Learning Abstraction Selection for Bayesian Program Analysis

YIFAN ZHANG, Peking University, China
YUANFENG SHI, Peking University, China
XIN ZHANG*, Peking University, China

We propose a learning-based approach to select abstractions for Bayesian program analysis. Bayesian program analysis converts a program analysis into a Bayesian model by attaching probabilities to analysis rules. It computes probabilities of analysis results and can update them by learning from user feedback, test runs, and other information. Its abstraction heavily affects how well it learns from such information. There exists a long line of works in selecting abstractions for conventional program analysis but they are not effective for Bayesian program analysis. This is because they do not optimize for generalization ability. We propose a data-driven framework to solve this problem by learning from labeled programs. Starting from an abstraction, it decides how to change the abstraction based on analysis derivations. To be general, it considers graph properties of analysis derivations; to be effective, it considers the derivations before and after changing the abstraction. We demonstrate the effectiveness of our approach using a datarace analysis and a thread-escape analysis.

CCS Concepts: • **Software and its engineering → Automated static analysis**; • **Mathematics of computing → Bayesian networks**; • **Information systems → Probabilistic retrieval models**; • **Computing methodologies → Machine learning approaches**.

Additional Key Words and Phrases: Static analysis, Bayesian network, alarm ranking, machine learning for program analysis, abstract interpretation

ACM Reference Format:
Yifan Zhang, Yuanfeng Shi, and Xin Zhang. 2024. Learning Abstraction Selection for Bayesian Program Analysis. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 128 (April 2024), 29 pages. https://doi.org/10.1145/3649845

1 INTRODUCTION

Abstract-interpretation-based program analyses [Cousot 1996] typically make over-approximations and are often expressed in logical rules. This can lead to a large number of false alarms in their results, which has a great negative impact on users' experience. Recently, a new paradigm which converts conventional analyses into Bayesian models was proposed to address this problem [Mangal et al. 2015]. We refer to it as Bayesian program analysis in the paper. In this paradigm, probabilities are attached to analysis rules to quantify their degrees of approximation. The generated reports also come with probabilities which are used to rank them. As a result, the analysis becomes a

*Corresponding author.

Authors' addresses: Yifan Zhang, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, yfzhang23@stu.pku.edu.cn; Yuanfeng Shi, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, friedrich22@stu.pku.edu.cn; Xin Zhang, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, xin@pku.edu.cn.

2475-1421/2024/4-ART128
https://doi.org/10.1145/3649845

Bayesian model and can improve its results by learning from various posterior information. Such information can come from user feedback [Mangal et al. 2015; Raghothaman et al. 2018], older versions of the program [Heo et al. 2019b], and test runs [Chen et al. 2021].

It is well-known that the choice of abstraction is crucial in balancing the trade-off between precision and scalability of conventional program analysis. The same problem applies to Bayesian program analysis. However, since a Bayesian program analysis is also a learning system, the choice of abstraction additionally affects how well the analysis generalizes from posterior information. Using a too fine abstraction may prevent posterior information from propagating to relevant analysis results effectively. On the other hand, using a too coarse abstraction may cause posterior information to propagate to irrelevant analysis results falsely. Since posterior information like user labels can be expensive to obtain, often choosing an abstraction that is optimized for generalization is more important than optimizing for precision and scalability. In other words, we want to choose an abstraction that can produce good alarm rankings with given amounts of posterior information.

In this paper, we aim to address such a challenge. Concretely, taking learning from user feedback as an example, our goal is to optimize the quality of the alarm ranking with the same amount of user feedback. However, we face two major challenges to solve this problem: (1) *Effectiveness*. Although there is a long line of works [Bielik et al. 2017; Grigore and Yang 2016; Hassanshahi et al. 2017; He et al. 2020; Heo et al. 2016, 2019a, 2017; Jeon et al. 2019, 2018, 2020; Jeon and Oh 2022; Jeong et al. 2017; Kastrinis and Smaragdakis 2013; Li et al. 2022, 2018a,b, 2020; Liang and Naik 2011; Liang et al. 2011; Lu and Xue 2019; Oh et al. 2014, 2015; Peleg et al. 2016; Singh et al. 2018; Smaragdakis et al. 2014; Tan et al. 2021, 2016, 2017; Wei and Ryder 2015; Zhang et al. 2014, 2013] on how to select adequate abstractions for conventional analysis, they cannot apply to the setting of Bayesian program analysis. They typically rely on a key assumption: given infinite resources, the finer the abstraction is, the better the analysis result is. However, such an assumption breaks due to the problem of generalizing posterior information. (2) *Generality*. While it is possible to develop effective solutions for particular analysis instances, our goal is to develop a methodology that works well for a wide range of Bayesian analyses.

To address these two challenges, we propose a data-driven approach, BinGraph, which selects program abstractions based on general characteristics of analysis derivations. For a specific program analysis, given a set of training programs whose true alarms are given, BinGraph identifies abstractions that are optimal for generalization and tries to learn a strategy to select such abstractions based on analysis derivation characteristics. Then given a new program to analyze, BinGraph first runs the analysis with a certain abstraction (typically the coarsest). By observing the analysis derivation, it decides how to modify the analysis abstraction. Compared to existing data-driven approaches for conventional analyses [Jeon et al. 2019, 2018, 2020; Jeon and Oh 2022; Jeong et al. 2017; Oh et al. 2015], BinGraph also considers the analysis derivation after applying a candidate abstraction modification to maximize effectiveness. In this way, our approach can be more accurate in predicting the effect of using a certain abstraction. Our evaluation shows such a learning approach based on changes in analysis derivations incurred by alternating the abstraction is effective in optimizing for generalization. In terms of the generality challenge, the features that BinGraph considers are graph properties of the derivations which are analysis-agnostic. As long as we can extract derivations graphs of a given analysis, BinGraph can apply. This is true for all existing Bayesian program analyses as they rely on derivation graphs to perform probabilistic inference.

We have implemented BinGraph and evaluated it on the Bayesian program analysis framework Bingo [Raghothaman et al. 2018] using two representative analyses: a datarace analysis with 4^N possible abstractions and a thread-escape analysis with 2^N possible abstractions on a suite of 13 Java programs of size 55-529 KLOC, where N is the number of object allocation statements in both analyses. We compare BinGraph to three baselines: the coarsest abstraction Base-C, the most

```
1  public class Thread1 extends Thread{
2      public static T1 global1;
3      public static T5 global2;
4      public void run(){
5          Scanner s = new Scanner(System.in);
6          switch(s.nextInt()){
7              case 1:
8                  global1 = new T1(s.next());    // H1
9                  global1.objA = new T2();       // H2
10                 T2 a = global1.objA;
11                 a.id = s.nextInt();            // E1
12                 a.name = s.next();             // E2
13                 break;
14             case 2:
15                 global1 = new T1(s.next());    // H3
16                 global1.objB = new T3();       // H4
17                 T3 b = global1.objB;
18                 b.id = s.nextInt();            // E3
19                 b.name = s.next();             // E4
20                 break;
21             case 3:
22                 global1 = new T1(s.next());    // H5
23                 global1.objC = new T4();       // H6
24                 T4 c = global1.objC;
25                 c.id = s.nextInt();            // E5
26                 c.name = s.next();             // E6
27                 break;
28         }
29         global2 = new T5(s.nextInt());         // H7
30         global2.objD = new T6();               // H8
31         T6 d = global2.objD;
32         d.objE = new T7();                     // H9
33         T7 e = d.objE;
34         e.id = s.nextInt();                    // E7
35         T1 local1 = new T1();                  // H10
36         local1.id = s.nextInt();               // E8
37         T5 local2 = new T5();                  // H11
38         local2.id = s.nextInt();               // E9
39     }
40 }
41 public class Thread2 extends Thread{
42     public void run(){
43         T5 global2 = Thread1.global2;
44         T6 d = global2.objD;
45         T7 e = d.objE;
46         System.out.println(e.id);
47     }
48     public static void main(String[] args){
49         new Thread1().start();
50         new Thread2().start();
51     }
52 }
```

Fig. 1. Code fragment of an example Java program.

precise abstraction Base-P, and Base-R, which is a abstraction with randomly selected granularity. On average, BinGraph has 45.36%, 23.38%, and 45.64% lower inversion count (the number of pairs of a false alarm inspected by the user before a true alarm) than these baselines, respectively.

Contributions. This paper makes the following contributions:

(1) We propose a framework BinGraph for learning abstraction selection for Bayesian program analysis. BinGraph has a direct optimization effect on the generalization ability and is general to apply to Bayesian program analyses with different logical rules.
(2) We show the effectiveness of BinGraph on diverse analyses applied to a suite of real-world programs. BinGraph significantly improves the generalization ability of the Bayesian program analyses compared to baselines.

2 MOTIVATING EXAMPLE

This section will take a thread-escape analysis on the Java code fragment in Figure 1 as an example to explain our problem and key idea. It is synthetic code for illustration. The concrete members of those classes are not important. Please focus on the points-to relation between fields and objects. There are two subclasses of `Thread`. The `run` method of `Thread1` allocates several objects, and the static fields of `Thread1` point to some of these objects. The `run` method of `Thread2` operates on a static field of `Thread1` and outputs relevant information.

There are 9 statements that the user is concerned with, labeled with `E1` to `E9` in the comments. The user wants to know if the targets on which these statements operate are accessed by multiple

Input relations

$\mathsf{FH}(h)$:	A static field may point to h.
$\mathsf{HFH}(h_1, h_2)$:	A non-static field of h_1 may point to h_2.
$\mathsf{EH}(e, h)$:	The statement e may operate on a non-static field of h.
$\mathsf{HL}(h, l)$:	The abstraction level for h is l.
$\mathsf{HT}(h, t)$:	The class type of h is t.

Output relations

$\mathsf{HX}(h, x)$:	h is considered as x during analysis.
$\mathsf{FX}(x)$:	A static field may point to x.
$\mathsf{XFX}(x_1, x_2)$:	A non-static field of x_1 may point to x_2.
$\mathsf{EX}(e, x)$:	The statement e may operate on a non-static field of x.
$\mathsf{escX}(x)$:	x may be accessed by multiple threads.
$\mathsf{escE}(e)$:	The target of e may be accessed by multiple threads.

Derivation rules

R_1 : $\mathsf{HX}(h, t)$:- $\mathsf{HL}(h, 0), \mathsf{HT}(h, t)$.
R_2 : $\mathsf{HX}(h, h)$:- $\mathsf{HL}(h, 1)$
R_3 : $\mathsf{FX}(x)$:- $\mathsf{FH}(h), \mathsf{HX}(h, x)$.
R_4 : $\mathsf{XFX}(x_1, x_2)$:- $\mathsf{HFH}(h_1, h_2), \mathsf{HX}(h_1, x_1), \mathsf{HX}(h_2, x_2)$.
R_5 : $\mathsf{EX}(e, x)$:- $\mathsf{EH}(e, h), \mathsf{HX}(h, x)$.
R_6 : $\mathsf{escX}(x)$:- $\mathsf{FX}(x)$.
R_7 : $\mathsf{escX}(x_2)$:- $\mathsf{escX}(x_1), \mathsf{XFX}(x_1, x_2)$.
R_8 : $\mathsf{escE}(e)$:- $\mathsf{EX}(e, x), \mathsf{escX}(x)$.

Fig. 2. A simplified parametric thread-escape analysis in Datalog. Here h, h_1, h_2 are allocation-site-based objects and x, x_1, x_2 are objects based on allocation sites or class types.

threads.* Since `Thread2` only operates the static field `global2`, only the operation target of statement `E7` is accessed by multiple threads during the actual execution of the program. We will show how a Bayesian parametric thread-escape analysis can help the user find statement `E7`.

2.1 A Parametric Thread-Escape Analysis

The analysis in Datalog is shown in Figure 2, which is simplified compared to the real analysis for exposition. The analysis is flow- and context-insensitive. Many analyses are parameterized to allow tuning their abstractions to balance the trade-off between precision and scalability. So is the example analysis. The parameters decide how to model various program facts in the abstraction, which we refer to as the *modeling strategies* for these facts. For example, in a cloning-based pointer analysis such as the k-object-sensitive pointer analysis [Milanova et al. 2005], each call site can be parameterized with a k value to decide the strategy to model the calling context associated with it. As for the thread-escape analysis, it is parameterized by how each heap object is modeled. There are two strategies to model an object: (1) it is considered as the same abstract object as other objects of the same class and the same modeling strategy, or (2) it is considered as the same abstract object as other objects that are created at the same line (allocation site) and of the same strategy. Two objects will not be considered as the same abstract object if they adopt different modeling strategies. There are 11 allocation sites related to the analysis, labeled with `H1` to `H11` in the comments. Their class types consist of `T1` to `T7`. For each allocation site, we can model the objects it allocates in one of these two strategies. If an allocation site adopts the first strategy, we

*Some thread-escape analyses concern whether certain objects are visible to multiple threads. Here, we care about accessibility which is more useful for downstream concurrency analyses such as datarace checkers.

say it has an abstraction level of 0, otherwise, the abstraction level is 1. Here, an *abstraction level* is the specific parameter to configure the modeling strategy for all objects created at an allocation site. In other analyses, abstraction levels may be associated with different program elements. In the example, we parameterize the whole abstraction used in the analysis by a Boolean vector for simplicity, whose length is the number of allocation sites in the program. The i-th element of the vector indicates the abstraction level of the i-th allocation site. When an allocation site adopts a higher abstraction level, the abstraction becomes more precise, but it may lead to less scalable. This parametric setup is used in conventional analysis to balance precision/scalability trade-offs.

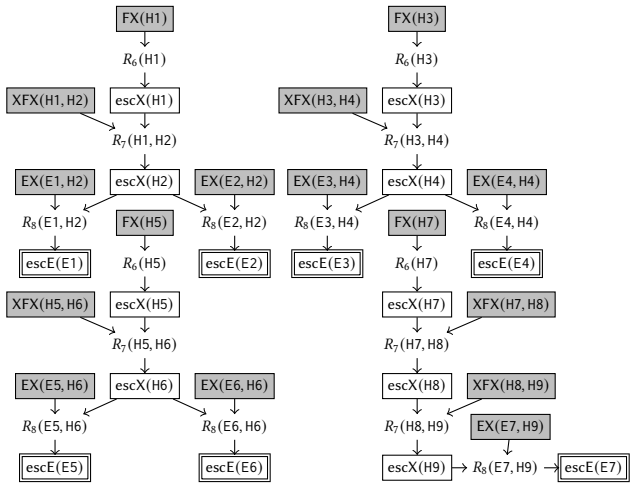
Concretely, relation HL encodes the abstraction, and HX encodes which abstract object that objects at an allocation site are modeled as. Rules R_1 and R_2 describe how to compute HX from HL. Besides HL, other input relations encode points-to information that is computed by an allocation-site-based pointer analysis. Rules R_3, R_4, and R_5 lift them to use the appropriate abstract objects based on HX. Finally, rules R_6, R_7 and R_8 describe the main logic of the analysis: (1) if an object is assigned to a static field, it escapes;[†] (2) if the field of an escaped object points to an object, the latter object also escapes. A Datalog inference engine takes these rules and the input relations (from the result of earlier analyses of the Java code fragment), and keeps deriving output tuples until no more output tuples can be derived. The rules over-approximate and can produce false alarms. The approximations come from (1) heap abstraction where multiple concrete objects at runtime are abstracted as one abstract object, and (2) the fact that if an object is reachable from a static field, it may not be accessed by multiple threads (e.g., the program is single-threaded). We explain (1) more using an example. Consider an abstract object O which consists of two concrete objects o_1 and o_2. Suppose o_1 is pointed by a static field, and o_2 does not escape. Then according to R_6, O escapes which includes o_2, which over-approximates. Suppose instruction e only accesses o_2, then according to R_8, e accesses an escaped object, which again over-approximates. The argument holds similarly for R_7.

When an abstraction is given, we can visualize all the input tuples, derived tuples, and the ground clauses (i.e., rule instances) involved in deriving these tuples as a directed graph. We refer to such tuples and rule instances as the *analysis derivation* for an analysis run, and the corresponding graph as the *derivation graph*. Figure 3 shows the derivation graphs under three different abstractions. In these graphs, vertices that are not wrapped in boxes represent relevant ground clauses. For example, R_7(H1, H2) represents one instance of rule R_7 involving elements H1 and H2. Vertices that are wrapped in boxes represent tuples. The ones with white backgrounds represent derived tuples while the ones with grey backgrounds represent input tuples. In particular, vertices with double frames represent alarm tuples. For each abstraction, the analysis gives 7, 9, and 8 alarms respectively, of which only escE(E7) is a true alarm. The more precise the abstraction, the fewer false alarms the analysis based on it will generate. However, it is difficult for the user to find the true alarm quickly even using the most precise abstraction. We next show how Bayesian program analysis helps the user find the true alarm faster.

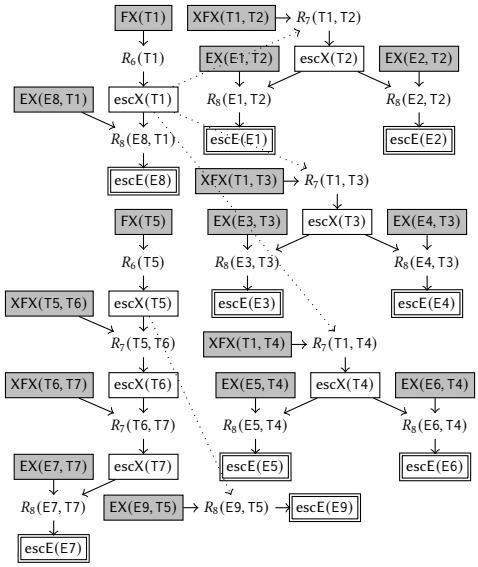
2.2 A Parametric Bayesian Program Analysis and the Abstraction Selection Problem

We first introduce Bayesian program analysis briefly. It transforms the analysis derivation to a probabilistic model, incorporates posterior information, calculates the probability of each alarm being true, and displays the highest one to the user. The user checks whether this alarm is true and feeds it back. Then, using the feedback as posterior information, the probability is calculated again and the interaction continues. We refer to the process of updating probabilities of alarms based on posterior information as *generalization*, and the ability to produce good alarm rankings

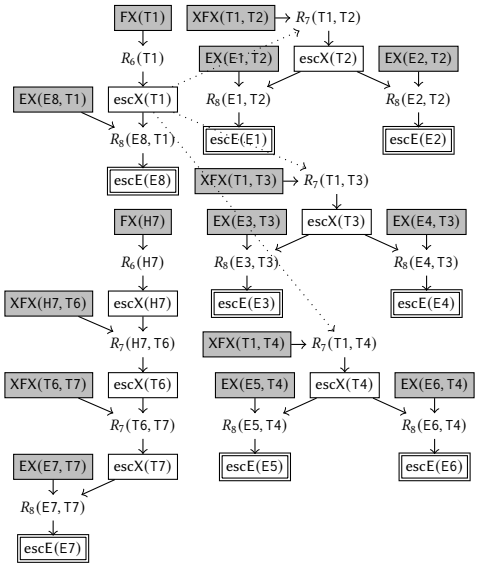
[†]Thread objects also escape. For simplicity, we do not consider them in the example.



(a) Abstraction S_1, represented as 11111111111.



(b) Abstraction S_2, represented as 00000000000.



(c) Abstraction S_3, represented as 00000010000

Fig. 3. The derivation graphs of the analysis in Figure 2 applying to the code fragment in Figure 1 under the three abstractions. Since the origin derivation graph is too huge to display, only derivation rules R_6 to R_8 are displayed and output relations of derivation rules R_3 to R_5 are treated as input relations. The dotted edges are just for display purposes and are not different from other edges.

with a given amount of posterior information as the *generalization ability* of a Bayesian program analysis. In the scenario of interactive alarm resolution, the stronger the generalization ability is, the more true alarms will be ranked first with the same amount of user feedback, or the less user

Table 1. The probability of each alarm before and after user feedback based on the abstraction S_1.

(a) $\Pr(x)$

Rank	Prob.	Alarm
1	0.857	escE(E1)
1	0.857	escE(E2)
1	0.857	escE(E3)
1	0.857	escE(E4)
1	0.857	escE(E5)
1	0.857	escE(E6)
7	0.815	escE(E7)

(b) $\Pr(x \mid \neg\text{escE(E1)})$

Rank	Prob.	Alarm
1	0.857	escE(E3)
1	0.857	escE(E4)
1	0.857	escE(E5)
1	0.857	escE(E6)
5	0.815	escE(E7)
6	0.301	escE(E2)
7	0	escE(E1)

feedback is needed to identify all true alarms. To estimate the generalization ability of Bayesian program analysis in the paper, we assume that the interaction does not stop until all true alarms have been checked, and in practice, the user may stop at any point they want. In the example, we use the number of rounds for the user to check all true alarms in the worst case (since there may be some alarms with equal probability) as an evaluation metric. A lower value indicates a better generalization ability.

Back to the example, rules R_6, R_7, R_8 over-approximate and may derive spurious program facts. We can quantify their imprecision by attaching probabilities to them. For simplicity, we set the probabilities to 0.95. In practice, the probabilities can be learned from labeled programs. Following this, a derivation graph can be converted into a Bayesian network. Specifically, each tuple and relevant ground clause in the derivation graph is considered as a Bernoulli random variable, representing whether the tuple or relevant ground clause holds. The relationships between adjacent vertices on the derivation graph are expressed using conditional probabilities. Taking relevant ground clause $R_7(\mathsf{H1},\mathsf{H2})$: $\mathsf{escX(H2)}$:- $\mathsf{escX(H1)}, \mathsf{XFX(H1,H2)}$ as an example, the corresponding conditional probabilities are:

$$\Pr(R_7(\mathsf{H1},\mathsf{H2}) \mid \mathsf{escX(H1)} \wedge \mathsf{XFX(H1,H2)}) = 0.95 \qquad \Pr(\mathsf{escX(H2)} \mid R_7(\mathsf{H1},\mathsf{H2})) = 1$$
$$\Pr(R_7(\mathsf{H1},\mathsf{H2}) \mid \neg\mathsf{escX(H1)} \vee \neg\mathsf{XFX(H1,H2)}) = 0 \qquad \Pr(\mathsf{escX(H2)} \mid \neg R_7(\mathsf{H1},\mathsf{H2})) = 0$$

Similar conditional probabilities are used for each relevant ground clause and its adjacent vertices on the derivation graph.

The probabilities that each alarm is true can be calculated by performing marginal inference on the Bayesian networks [Murphy et al. 1999]. They are used to rank the alarms. We take the finest abstraction S_1 as an example. The probability of each alarm being true is shown in Table 1a. The analysis displays the most probable alarm, escE(E1), to the user. The user finds that this alarm is false and gives negative feedback. The analysis considers it as posterior information and updates the probability of each alarm as shown in Table 1b. The interaction continues as the user pleases. As shown in Table 2, in the 4-th round, the user receives the true alarm. Note that even though there can be multiple most probable alarms in each round, the user always only needs to inspect 4 alarms to find the true alarm no matter which alarm is posed. On the other hand, the user needs to inspect all 7 alarms in the worst case using the conventional analysis.

The performance of the Bayesian analysis can be further boosted using a better abstraction. Consider abstraction S_3 where only the abstraction level of H7 that is directly related to the true alarm is assigned to 1. Using this abstraction, the performance of the conventional analysis degrades as it now derives 8 alarms. However, using the Bayesian analysis, the user only needs to inspect two alarms to find the true alarm. The reason is that a more coarse abstraction can sometimes

Table 2. The alarm with the highest probability in each round based on each abstraction. Statistics after the 4-th round are not presented and they are all false alarms.

Round	S_1 = 11111111111		S_2 = 00000000000		S_3 = 00000010000	
	Prob.	**Alarm**	**Prob.**	**Alarm**	**Prob.**	**Alarm**
1	0.857	escE(E1)	0.903	escE(E8)	0.903	escE(E8)
2	0.857	escE(E3)	0.903	escE(E9)	0.815	escE(E7)
3	0.857	escE(E5)	0.440	escE(E1)	0.440	escE(E1)
4	0.815	escE(E7)	0.418	escE(E7)	0.077	escE(E3)
...	...	...	...	...	...	...

correlate more false alarms together. As a result, providing feedback on a false alarm can generalize to more false alarms. Let us take a closer look at the derivation graph under S_3 in Figure 3c. In it, all false alarms are connected but disconnected from the only true alarm. As a result, providing negative feedback on any false alarm will decrease the probability of all the other false alarms. But the probability of the true alarm is unaffected and therefore its rank is improved. On the other hand, in the derivation graph under S_1, most false alarms are disconnected.

Using a too coarse-grained abstraction can also be harmful. Let us consider the cheapest abstraction S_2. In its derivation graph shown in Figure 3b, all alarms are connected together. So feedback on negative alarms will also affect the probability of the true alarm. As a result, its performance is worse than that of S_3.

From the observations, we draw two key insights. First, the abstraction selection heavily affects the Bayesian analysis' performance. Second, due to the problem of generalization, the Bayesian analysis' performance does not align with that of the conventional analysis. For conventional analyses, ignoring efficiency, more precise abstractions will not lead to worse results. However, in this example, S_3 which is an abstraction in the middle in terms of precision, is the optimum abstraction for the Bayesian analysis in all 2^{11} possible abstractions. This shows that the abstraction selection problem of Bayesian analysis is fundamentally different from that of conventional analysis, and we need new techniques.

2.3 Our Approach

Figure 4 shows the workflow of our approach to this problem, BinGraph. BinGraph is a learning-based approach. Let us focus on its online part for now. Starting from the coarsest abstraction (S_2 in the example), it tries to refine the abstraction iteratively. Although in the example the abstraction level can only be a Boolean value, in general, it can be a natural number. Further, the parameterization may not be associated with allocation sites in other analyses, but with program elements such as methods and variables. We refer to these program elements as *abstraction points*. Raising the abstraction level for abstraction points makes the abstraction more precise. This is a very common setting in conventional analysis. For example, the abstraction level is the degree of context-sensitivity for each call site in parametric k-object-sensitive pointer analysis [Milanova et al. 2005]. Moreover, for each abstraction point with a certain abstraction level, a parameter tuple is included in the input. For example, if the abstraction level for H1 is 0, then the parameter tuple is HL(H1, 0). If the abstraction level for H1 is 1, then the parameter tuple is HL(H1, 1).

In each iteration, BinGraph decides to raise the abstraction level of each abstraction point by one or keep it unchanged. The number of iterations is the same as the maximum abstraction level (only one in the example). As a learning-based approach, BinGraph characterizes the impact of

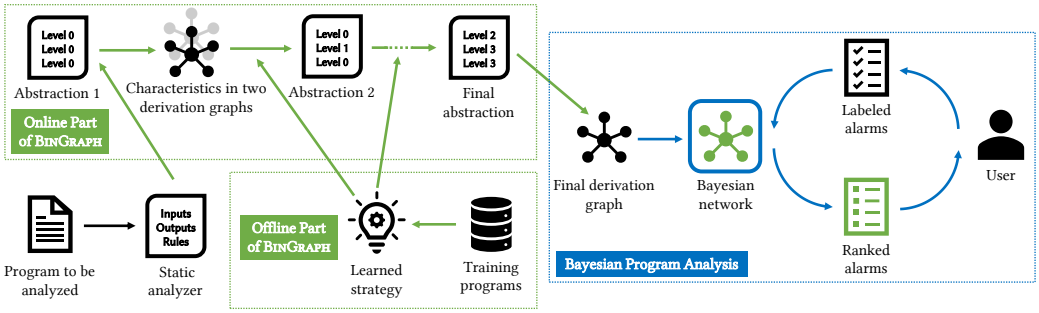


Fig. 4. Overview of framework BinGraph for learning abstraction selection for Bayesian program analysis.

raising the level for each abstraction point, then determines whether to raise it based on the learned strategy.

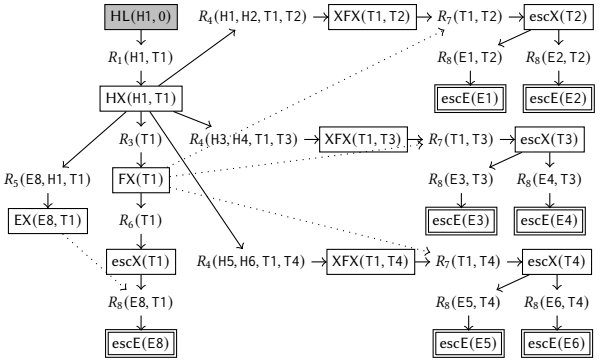
To obtain *Effectiveness*, BinGraph considers two derivation graphs when characterizing a parameter. The first one is the derivation graph under the current abstraction (S_2 in the example). Another one is the derivation graph under the abstraction after overall refinement (i.e., raising the abstraction level for each abstraction point by one, corresponding to S_1 in the example). The analyzed information under a coarse abstraction may reflect relevant properties when using a more precise abstraction in conventional analysis. This is mainly based on that the finer the abstraction is, the better the analysis result is. Since this assumption does not hold for Bayesian analysis, only using information under a coarse abstraction is unable to **accurately** predict the effect of analysis under a more precise abstraction. Therefore, using both derivation graphs under two abstractions **ensures** that the abstraction is made more beneficial to analysis results in each iteration.

To obtain *Generality*, BinGraph only considers derivation graph properties that are related to parameter tuples ($\mathrm{HL}(x, y)$) in the example). While BinGraph can be configured with different graph properties, given a parameter tuple, it uses three property types in the experiment: (1) the count of reachable vertices, (2) the average of shortest distance to reachable vertices, and (3) the count of vertices with shortest distance $\leq k$. Note that these properties are only relevant to the derivation graph and not to the semantics of the analysis, so they can be applied to any type of Bayesian program analysis. Moreover, adding other property types is also supported in BinGraph. The intuition behind choosing these three kinds of features is that they reflect the potential impact of refining parameter tuples on information propagation in a Bayesian network:

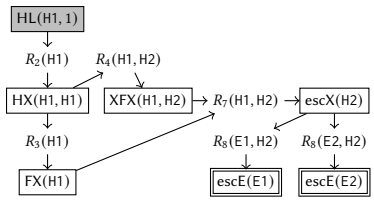
(1) **The count of reachable vertices.** It reflects the number of vertices that are potentially affected by refining a parameter tuple. In other words, it reflects the overall influence of refining the tuple on the Bayesian network.
(2) **The average of shortest distance to reachable vertices.** Since the impact on each vertex in the Bayesian network becomes weaker when the distance from the parameter tuple becomes farther, this feature reflects the average impact on reachable vertices of refining the parameter tuple.
(3) **The count of vertices with shortest distance $\leq k$.** This feature reflects the potential influence to a certain subgraph. In other words, it reflects the number of affected vertices within a certain distance. Moreover, irrelevant to information propagation, this feature can capture different subgraph patterns within a given radius, which in turn can be used to

Table 3. Properties in two derivation graphs. Note that properties are a subset of those used in experiments, and the types and number of properties can be arbitrarily set in actual use.

Property type	HL(H1, 0)	HL(H1, 1)	Ratio
The count of reachable vertices	35	13	0.371
The average of shortest distance to reachable vertices	4.114	4.462	1.085
The count of vertices with shortest distance ≤ 5	18	8	0.444



(a) The derivation graph based on S_2 and the vertex HL(H1,0).



(b) The derivation graph based on S_1 and the vertex HL(H1,1).

Fig. 5. The derivation graphs based on S_2 and S_1. Only reachable vertices of parameter tuples are displayed.

identify parameter tuples that need to be refined. In our experiment, we have ten features of this kind where $k = 1, 2, \ldots, 10$.

We did not design these features specifically for the analyses in the paper but believe they are general features that reflect the impact of information propagation. However, there is space for carefully engineering features for a specific analysis to achieve even better performance.

We take the part related to H1 as an example to explain BinGraph. BinGraph calculates the properties of HL(H1, 0) in the derivation graph under S_2 and the properties of HL(H1, 1) in graph under S_1. Here, we consider three properties which are shown in Table 3, where only one distance threshold is considered for the third property for simplicity. Figure 5 shows the subgraphs that are relevant to compute these properties. The actual feature vector is a three-dimensional real vector, each element of which is a ratio between a property on the original graph and its counterpart on the refined graph. Table 3 shows these values for H1. Following this approach, H3 and H5 have the same characteristic $(0.371, 1.085, 0.444)$ as H1, while H7 has the different characteristic $(0.882, 0.940, 0.778)$. The strategy learned by BinGraph is represented as a set of three-dimensional cubes: only parameter tuples whose feature values fall into them will be refined. For example, suppose, by training on similar programs, BinGraph learns a strategy that is represented by one cube $[0.8, 1] \times [0.9, 1] \times [0.7, 1]$. Using this strategy, BinGraph is able to separate H7 from other allocations. Its allocation site is precisely what we need to raise the abstraction level to obtain the optimum abstraction S_3. We will go into more detail about the online and offline parts of BinGraph in Section 4.

$$
\begin{array}{rl}
(\textit{variables}) & \mathbf{V} = \{h_1, x_2, \dots\} \\
(\textit{constants}) & \mathbf{D} = \{\mathsf{E1}, \mathsf{H1}, 0, 1, \dots\} \\
(\textit{relations}) & \mathbf{R} = \{\mathsf{HFH}, \mathsf{escE}, \dots\} \\
(\textit{literals}) & \mathbf{L} = \mathbf{R} \times (\mathbf{D} \cup \mathbf{V})^* = \{\mathsf{HFH}(h_1, h_2), \mathsf{HL}(h, 0), \dots\} \\
(\textit{tuples}) & \mathbf{T} = \mathbf{R} \times \mathbf{D}^* = \{\mathsf{HFH}(\mathsf{H1}, \mathsf{H2}), \mathsf{HL}(\mathsf{H1}, 0), \dots\} \\
(\textit{clauses}) & \mathbf{C} = \mathbf{L} \times \mathbf{L}^* = \{[\mathsf{escX}(x_2) \text{ :- } \mathsf{escX}(x_1), \mathsf{XFX}(x_1, x_2)], \dots\}
\end{array}
$$

Fig. 6. Auxiliary definitions and notations of Datalog.

$$
\begin{array}{c}
F_R, f_c \in \mathscr{P}(\mathbf{T}) \to \mathscr{P}(\mathbf{T}) \\
F_R(T) = T \cup \{f_c(T) \mid c \in R\} \\
f_{[l_0 \text{:-} l_1, \dots, l_n]}(T) = \{\sigma(l_0) \mid \sigma(l_i) \in T \text{ for } 1 \le i \le n, \sigma \in \Sigma\}
\end{array}
$$

Fig. 7. Semantics of Datalog.

3 PRELIMINARIES

3.1 Datalog Syntax and Semantics

A Datalog program $\mathcal{D} = (I, O, R)$ consists of input relations $I \subseteq \mathbf{R}$, output relations $O \subseteq \mathbf{R}$ and derivation rules $R \subseteq \mathbf{C}$. The auxiliary definitions and notations are shown in Figure 6. A *substitution function* $\sigma \in \Sigma = \mathbf{V} \to \mathbf{D}$ replaces a variable with a constant. We also abuse its notation so it applies to a literal by replacing all variables in the literal with constants according to the function. In other words, for a literal $l = r(a_1, a_2, \dots, a_n)$, $\sigma(l) = r(b_1, b_2, \dots, b_n)$ where $b_i = a_i$ if $a_i \in \mathbf{D}$ else $b_i = \sigma(a_i)$. The output tuples of the Datalog program $\mathcal{D}$ with the input tuples $T_0 \subseteq \mathbf{T}$ is defined as $[\![\mathcal{D}, T_0]\!] = \mathsf{lfp}(F_R, T_0)$. Here, F_R computes output tuples by applying rules in R to a given set of tuples for one round. In other words, let T be the output tuples a Datalog program computes: the program initially makes $T \leftarrow T_0$ and then keeps making $T \leftarrow F_R(T)$ until $T = F_R(T)$, at which point T is $[\![\mathcal{D}, T_0]\!]$. Figure 7 shows the relevant definitions.

3.2 Parametric Datalog Program Analysis

We now turn to Datalog programs that implement parametric program analyses. Compared to a standard Datalog program, its input now consists of two parts: (1) a set of tuples that are extracted from a given program $P \in \mathbf{P}$, (2) a set of tuples that encode abstraction parameters. For simplicity, we omit the first part and assume a program P is given. While the form of the abstraction family varies, the specific form we consider is parameterized by an array of natural numbers. Such a form can encode rich ways to parameterize an analysis, including context sensitivity, lengths of access paths, number of unrollings of a loop, and others. Typically, the length of such an array varies across programs and each element is associated with a program fact such as an allocation site. We refer to such program facts as *abstraction points*, and the set of all abstraction points in a given program P as an abstraction point set $AS \subset \mathbf{D}$. We refer to the number associated with an abstraction point as its abstraction level. We use AL to denote the allowed maximum abstraction level. The *abstraction*, or the analysis's configuration is defined as $S \in AS \to \{0, 1, \dots, AL\}$. So there are total $(AL + 1)^{|AS|}$ possible abstractions. Specifically, we define $\mathcal{S}_0$ as the coarsest abstraction such that $\mathcal{S}_0(x) = 0$ holds for $x \in AS$. We generate parameter tuples using function $AI \in \mathbf{D} \times \mathbb{N} \to \mathbf{T}$ which maps an abstraction point and its level to a tuple. The output tuples based on the abstraction S are defined as $\textsc{Output}(S) = [\![\mathcal{D}, \{AI(x, S(x)) \mid x \in AS\}]\!]$.

Among the output tuples, we use $q \in \mathbf{R}$ to denote a *query relation* that represents alarms. The output alarms is defined as $\textsc{Alarms}(S) = \{t \mid t = q(a_1, a_2, \ldots, a_n) \in \textsc{Output}(S)]\}$. Typically, a higher abstraction level leads to a more precise and expensive abstraction. Formally, for a program P, abstractions S and S', if $S(x) \leq S'(x)$ holds for any $x \in AS$, then $\textsc{Alarms}(S) \supseteq \textsc{Alarms}(S')$.

As a result, a parametric Datalog program analysis can defined as $\mathcal{A} = (\mathcal{D}, q, AS, AL, AI)$.

Example 3.1. Consider the parametric Datalog analysis $\mathcal{A} = (\mathcal{D}, q, AS, AL, AI)$ shown in Section 2 applying to the program P shown in Figure 1: $\mathcal{D}$ is shown in Figure 2; q is the relation escE; AS is the set of allocation sites $\{\mathsf{H1}, \mathsf{H2}, \ldots, \mathsf{H11}\}$; AL is equal to 1; $AI(x, y) = \mathsf{HL}(x, y)$ where $x \in AS$ and $y \in \{0, 1\}$; tuples in FH, HFH, and EH are the non-parameter input tuples. For three abstractions we presented in the example, $S_1(x) = 1$ holds for $x \in AS$, $S_2 = \mathcal{S}_0$, $S_3(x) = 0$ holds for $x \in AS - \{\mathsf{H7}\}$ and $S_3(\mathsf{H7}) = 1$. Taking S_1 as an example, $\textsc{Output}(S_1) = \{\mathsf{FX}(\mathsf{H1}), \mathsf{XFX}(\mathsf{H1}, \mathsf{H2}), \mathsf{escX}(\mathsf{H1}), \mathsf{escE}(\mathsf{E1}), \ldots\}$ is the set of all tuples in output relations HX, FX, XFX, EX, escX and escE. $\textsc{Alarm}(S_1) = \{\mathsf{escE}(\mathsf{E1}), \mathsf{escE}(\mathsf{E2}), \ldots, \mathsf{escE}(\mathsf{E7})\}$ is the set of tuples representing alarms. Since $S_2(x) \leq S_3(x) \leq S_1(x)$ holds for any $x \in AS(P)$, $\textsc{Alarms}(S_2) \supseteq \textsc{Alarms}(S_3) \supseteq \textsc{Alarms}(S_1)$ holds.

3.3 Parametric Bayesian Program Analysis

We only introduce the Bayesian program analysis using user feedback as posterior information in this part. A parametric Bayesian program analysis is based on a parametric Datalog program analysis $\mathcal{A} = (\mathcal{D}, q, AS, AL, AI)$. We assume abstraction S is used when analyzing program P. We next explain how to convert a Bayesian analysis into a probabilistic graphical model to compute the probabilities of the alarms. A *ground clause* is a clause that does not involve any variable. We refer to a ground clause that is involved in the analysis derivation as a *relevant ground clause*. Formally, a relevant ground clause is $([l_0 \text{ :- } l_1, \ldots, l_n], t_0, t_1, \ldots, t_n) \in \mathbf{C} \times \mathbf{T}^*$ such that it satisfies $t_i \in \textsc{Output}(S)$ and there exists a function $\sigma \in \Sigma$ that $\sigma(l_i) = t_i$ holds. The set of all relevant ground clauses is defined as $\textsc{Ground}(S)$. The *derivation graph* is defined as $\textsc{Graph}(S) = (V, E)$. It is a directed graph and $V = \textsc{Output}(S) \cup \textsc{Ground}(S)$. For each $i = (c, t_0, t_1, \ldots, t_n) \in \textsc{Ground}(S)$, there exist $n + 1$ directed edges $(t_1, i), \ldots, (t_n, i), (i, t_0)$ in E. A derivation graph with the above definition may contain cycles, which can be problematical for efficient inference. Following previous works [Chen et al. 2021; Heo et al. 2019b; Kim et al. 2022; Raghothaman et al. 2018], we remove cycles in the graph. In the rest of the paper, we assume $\textsc{Graph}(S) = (V, E)$ contains no cycles.

Example 3.2. Take the parametric Datalog analysis $\mathcal{A} = (\mathcal{D}, q, AS, AL, AI)$ shown in Section 2 using abstraction S_1 applying to the program P shown in Figure 1 as an example: $([\mathsf{escX}(x_2) \text{ :- } \mathsf{escX}(x_1), \mathsf{XFX}(x_1, x_2)], \mathsf{escX}(\mathsf{H2}), \mathsf{escX}(\mathsf{H1}), \mathsf{XFX}(\mathsf{H1}, \mathsf{H2})) \in \textsc{Ground}(S_1)$, and the vertex representing it in the derivation graph $\textsc{Graph}(S_1)$ is $R_7(\mathsf{H1}, \mathsf{H2})$ in Figure 3a. Note that the three derivation graphs in Figure 3 are not complete for display convenience.

To compute the marginal probabilities of the alarms, the derivation graph is compiled into a Bayesian network. For each v in V, a Bernoulli random variable x_v is created. We denote the set of these random variables as X. In addition, we assume there exists a function $Y \in \mathbf{C} \to [0, 1]$ that assigns a probability to each rule in the original Datalog analysis. Such a function can be learned on labeled data or specified by experts. Since our focus is the impact of abstraction selection, we use the same configuration in previous research [Raghothaman et al. 2018] where we set $Y(c) = 0.999$ for all rules in the experiments. For each $t \in \textsc{Output}(S)$, let the relevant ground rules that can derive it be $i_1, i_2, \ldots, i_n$, we create edges between their corresponding random variables with conditional probabilities $\Pr(x_t \mid x_{i_1} \vee x_{i_2} \vee \cdots \vee x_{i_n}) = 1$ and $\Pr(x_t \mid \neg x_{i_1} \wedge \neg x_{i_2} \wedge \cdots \wedge \neg x_{i_n}) = 0$. For each $i = (c, t_0, t_1, \ldots, t_n) \in \textsc{Ground}(S)$, we create edges with conditional probabilities $\Pr(x_i \mid$

$x_{t_1} \wedge x_{t_2} \wedge \cdots \wedge x_{t_n}) = Y(c)$ and $\Pr(x_i \mid \neg x_{t_1} \vee \neg x_{t_2} \vee \cdots \vee \neg x_{t_n}) = 0$. Then, $B = (V, E, X, Y)$ forms a Bayesian network [Koller and Friedman 2009].

Example 3.3. Consider the following two relevant ground clauses:

$$c_1 = ([\mathsf{A}(t) \text{ :- } \mathsf{B}(t), \mathsf{C}(t)], \mathsf{A}(t_1), \mathsf{B}(t_1), \mathsf{C}(t_1)) \quad c_2 = ([\mathsf{A}(t) \text{ :- } \mathsf{D}(t), \mathsf{E}(t)], \mathsf{A}(t_1), \mathsf{D}(t_1), \mathsf{E}(t_1))$$

The corresponding conditional probabilities in the Bayesian network are:

$$\begin{aligned}
&\Pr(x_{c_1} \mid x_{\mathsf{B}(t_1)} \wedge x_{\mathsf{C}(t_1)}) = 0.999 && \Pr(x_{c_1} \mid \neg x_{\mathsf{B}(t_1)} \vee \neg x_{\mathsf{C}(t_1)}) = 0 \\
&\Pr(x_{c_2} \mid x_{\mathsf{D}(t_1)} \wedge x_{\mathsf{E}(t_1)}) = 0.999 && \Pr(x_{c_2} \mid \neg x_{\mathsf{D}(t_1)} \vee \neg x_{\mathsf{E}(t_1)}) = 0 \\
&\Pr(x_{\mathsf{A}(t_1)} \mid x_{c_1} \vee x_{c_2}) = 1 && \Pr(x_{\mathsf{A}(t_1)} \mid \neg x_{c_1} \wedge \neg x_{c_2}) = 0
\end{aligned}$$

Finally, we present using a Bayesian program analysis to perform interactive alarm resolution. The interaction consists of multiple rounds, in each of which the analysis produces an alarm and the user inspects it and provides binary feedback. Let E_i be the set of user feedback before the i-th round. Initially, $E_1 = \emptyset$. For the i-th round, using probability inference algorithms [Murphy et al. 1999] on the Bayesian network B, the alarm with the highest probability, $a = \arg\max_{t \in \textsc{Alarm}(S)} \Pr\left(x_t \mid \bigwedge_{e \in E_i} e\right)$, will be displayed to the user. The user will check if a is true and feed it back. If a is true then $E_{i+1} = E_i \cup \{x_a\}$ else $E_{i+1} = E_i \cup \{\neg x_a\}$, and the interaction moves to the next round. The user may terminate the interaction at any time. To evaluate the generalization ability of Bayesian program analysis in experiments, we assume that the user do not terminate until all alarms have been checked. This setup follows recent works [Chen et al. 2021; Heo et al. 2019b; Kim et al. 2022; Raghothaman et al. 2018] in Bayesian program analysis. In practice, the user may use other termination conditions. For example, they may decide to stop after n consecutive alarms are false. In our experiment, it is usually that a small fraction of the true alarms can only be discovered after inspecting many false alarms. Let l_i be 1 if the alarm displayed in the i-th round is true else be 0. We use the inversion count of $l_1, l_2, \ldots, l_n$ to evaluate the quality of the analysis results. The inversion count is the number of pairs of a false alarm and a true alarm such that the false alarm is inspected by the user before the true alarm, and thus reflects the generalization ability. We define $\textsc{Inversion}(S) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} [l_i > l_j]$[‡]. There are also three other metrics used in previous research. We will demonstrate them in Section 5 as a supplement.

Example 3.4. Take the parametric Datalog analysis $\mathcal{A} = (\mathcal{D}, q, AS, AL, AI)$ shown in Section 2 applying to the program P shown in Figure 1 as an example: $\textsc{Inversion}(S_1) = \textsc{Inversion}(S_2) = 3$ and $\textsc{Inversion}(S_3) = 1$, so the Bayesian program analysis under abstraction S_3 has stronger generalization ability according to our metric. Table 2 visualizes the whole process of interaction. The probabilities of the rules we presented in the example are 0.95, which are different from those in the experiments.

4 THE BINGRAPH FRAMEWORK

Given a parametric Bayesian program analysis based on a parametric Datalog program analysis $\mathcal{A} = (\mathcal{D}, q, AS, AL, AI)$, the goal of abstraction selection problem is to find a function $f \in \mathbf{P} \to (\mathbf{D} \to \mathbb{N})$ to minimize a given metric (e.g., $\textsc{Inversion}(f(P))$) for every program to be analyzed $P \in \mathbf{P}$. BinGraph applies a data-driven approach to address this problem and consists of two parts. The online part selects an abstraction for a given program by iteratively raising abstraction levels of abstraction points. The offline part learns a strategy from training programs for use in the online part. We will introduce these two parts in the next two subsections.

[‡][ ] denotes Iverson Bracket. If the statement S is true, then $[S] = 1$ else $[S] = 0$.

4.1 Online Part of BinGraph

We summarize the online selection process of BinGraph. As shown in Figure 4, starting from the coarsest abstraction $\mathcal{S}_0$, it iteratively refines the abstraction for AL rounds where AL is the maximum abstraction level. We use S_i to denote the abstraction after the i-th round, and $S_0 = \mathcal{S}_0$ to denote the initial abstraction. In the i-th round, BinGraph calculates analysis derivation characteristics for each abstraction point based on the abstraction S_{i-1}. For each abstraction point x, if its characteristics matches the learned strategy, then $S_i(x) \leftarrow S_{i-1}(x)+1$ else $S_i(x) \leftarrow S_{i-1}(x)$. After the AL-th round, S_{AL} will be the abstraction selected by BinGraph. Our design chooses to iterate a fixed number of AL times, instead of setting a termination condition such as $S = S'$ or when the difference between S and S' is small. This is because it can take many iterations for it to be satisfied. There are two main reasons: (1) there may be noise in our learned strategy, and (2) after refining an abstraction point, the features of other abstraction points can change, which may lead to a chain reaction consisting of many rounds. Our design is a simpler alternative which ensures that the highest abstraction level is reachable.

Example 4.1. Suppose the maximum abstraction level $AL = 3$ and the size of the set of abstraction points $|AS| = 5$. We use a natural number vector of length 5 to represent an abstraction for simplicity. The i-th element of the vector indicates the abstraction level of the i-th abstraction point. Initially, S_0 can be represented as 00000. In the 1-st round of the online part of BinGraph, the overall refinement of S_0 is S_0' : 11111. BinGraph characterizes each of 5 abstraction points based on two derivation graphs based on S_0 and S_0'. We assume only the characteristics of the 3-rd and the 4-th abstraction points match the learned strategy, then we have S_1 : 00110. In the 2-nd round, the overall refinement of S_1 is S_1' : 11221. BinGraph characterizes each of 5 abstraction points based on two derivation graphs based on S_1 and S_1'. We assume only the characteristics of the 3-rd and the 5-th abstraction points match the learned strategy, then we have S_2 : 00211. In the 3-rd round, the overall refinement of S_2 is S_2' : 11322. BinGraph characterizes each of 5 abstraction points based on two derivation graphs based on S_2 and S_2'. We assume only the characteristics of the 3-rd and the 4-th abstraction points match the learned strategy, then we have S_3 : 00321. Finally, S_3 is the abstraction selected by BinGraph.

We present the selection algorithm in Algorithm 1 and explain relevant definitions in detail. The analysis derivation characteristics are calculated based on an abstraction and its overall refinement. An *overall refinement* raises the abstraction levels of all abstraction points by one. Formally, given a program P and an abstraction S, the overall refinement of S is $S' \in AS \to \{0, 1, \ldots, AL\}$ such that $S'(x) = S(x) + 1$ for $x \in AS$. BinGraph uses graph properties to characterize each abstraction point on the derivation graphs based on abstraction S and S'. Formally, let $\mathcal{N}$ be the number of properties and $\beta_i(G, v) \in \mathbb{R}$ be the i-th property for the vertex v in graph G.

To combine characteristics of each abstraction point in two derivation graphs, we define the *feature value* for each abstraction point x as $\lambda(S, x) \in \mathbb{R}^{\mathcal{N}}$, where the i-th component is:

$$\lambda_i(S, x) = \frac{\beta_i(\textsc{Graph}(S'), AI(x, S'(x)))}{\beta_i(\textsc{Graph}(S), AI(x, S(x)))}$$

The feature value of an abstraction point represents its analysis derivation characteristics based on a certain abstraction. Feature values will be used throughout the process of BinGraph. The strategy *Learned* is a set of feature values calculated in the offline part. If the feature value of an abstraction point is contained in *Learned*, then its abstraction level is raised. The types and numbers of properties we use in experiments will be demonstrated in Section 5.

Example 4.2. Take the parametric Datalog analysis $\mathcal{A} = (\mathcal{D}, q, AS, AL, AI)$ shown in Section 2 applying to the program P shown in Figure 1 as an example: the types of properties are shown in Table 3

Algorithm 1 Selection algorithm

Input: A strategy $Learned \subseteq \mathbb{R}^{\mathcal{N}}$.
Output: A function $f \in \mathbf{P} \to (\mathbf{D} \to \mathbb{N})$.

procedure SELECT($Learned$)
  Let $P \in \mathbf{P}$ be the program to be analyzed, $S_0 \leftarrow \mathcal{S}_0$
  for $i = 1 \to AL$ **do**
    $S_i \leftarrow S_{i-1}$
    for $x \in AS(P)$ **do**
      $S_i(x) \leftarrow S_{i-1}(x) + [\lambda(S_{i-1}, x) \in Learned]^{\ddagger}$
  return $f(P) = S_{AL}$

Algorithm 2 Labeling algorithm

Input: A set of training programs $P_{\mathrm{T}} \subseteq \mathbf{P}$.
Output: Labeled sets of feature values $Labeled_0, Labeled_1 \subseteq \mathbb{R}^{\mathcal{N}}$.

procedure LABEL(P_{T})
  $Labeled_0 \leftarrow \emptyset, Labeled_1 \leftarrow \emptyset$
  for $P \in P_{\mathrm{T}}$ **do**
    Let P be the program to be analyzed, $S_0 \leftarrow \mathcal{S}_0$
    for $i = 1 \to AL$ **do**
      Using SA to find $S_i \in \textsc{Search}(S_{i-1})$ minimizing $\textsc{Inversion}(S_i)$
      for $x \in AS(P)$ **do**
        $l \leftarrow [S_i(x) \neq S_{i-1}(x)]^{\ddagger}$
        $Labeled_l \leftarrow Labeled_l \cup \{\lambda(S_{i-1}, x)\}$
  return $Labeled_0, Labeled_1$

with $\mathcal{N} = 3$. $\beta_1(\textsc{Graph}(S_2), AI(\mathsf{H1}, 0)) = 35$ represents the count of reachable vertices from vertex $\mathsf{HL}(\mathsf{H1}, 0)$ in the derivation graph shown in Figure 5a. It can be shown that the overall refinement of S_2 is S_1. $\beta_1(\textsc{Graph}(S_1), AI(\mathsf{H1}, 1)) = 13$ represents the count of reachable vertices from vertex $\mathsf{HL}(\mathsf{H1}, 1)$ in the derivation graph shown in Figure 5b. Therefore, $\lambda_1(S_2, \mathsf{H1}) = \frac{13}{35} = 0.371$. Similarly, $\lambda(S_2, \mathsf{H1}) = \lambda(S_2, \mathsf{H3}) = \lambda(S_2, \mathsf{H5}) = (0.371, 1.085, 0.444)$ and $\lambda(S_2, \mathsf{H7}) = (0.882, 0.940, 0.778)$.

4.2 Offline Part of BinGraph

The offline part of BinGraph consists of a labeling algorithm and a learning algorithm. For each training program, we obtain the true alarms on them in advance and simulate the whole interaction under different abstractions automatically to calculate inversion counts. Algorithm 2 shows the labeling algorithm, which generates two labeled sets of feature values $Labeled_0, Labeled_1$ for subsequent supervised learning. $Labeled_0$ corresponds to abstraction points whose abstraction levels should not be raised, while $Labeled_1$ corresponds to the opposite. The process of labeling each training program is similar to selection. It consists of AL rounds and maintains an abstraction S_i after the i-th round. The difference is that S_i is an optimal abstraction obtained by searching based on S_{i-1}. Formally, we define the search space in the i-th round as $\textsc{Search}(S_{i-1}) = \{S \mid S \in \hookrightarrow \{0, 1, \ldots, AL\}, S_{i-1}(x) \leq S(x) \leq S_{i-1}(x) + 1\}$. In the i-th round, an abstraction S_i with low $\textsc{Inversion}(S_i)$ is searched by Simulated Annealing (SA) [Kirkpatrick et al. 1983]. If the abstraction level of an abstraction point x is raised, then $\lambda(S_{i-1}, x)$ is labeled with 1 else 0. It is obvious that levels of abstraction points with similar characteristics to $Labeled_0$ should not be raised, and it is

Algorithm 3 Learning algorithm

Input: Labeled sets of feature values $Labeled_0, Labeled_1 \subseteq \mathbb{R}^{\mathcal{N}}$.
Output: A strategy $Learned \subseteq \mathbb{R}^{\mathcal{N}}$.

```
1: procedure LEARN(Labeled_0, Labeled_1)
2:     Learned ← ∅, L_0 ← Labeled_0, L_1 ← Labeled_1
3:     while L_1 ≠ ∅ do
4:         C ← FINDCUBE(L_0, L_1)
5:         Learned ← Learned ∪ C, L_0 ← L_0 − C, L_1 ← L_1 − C
6:     return Learned
```

the opposite for $Labeled_1$. The learning process of BINGRAPH is designed to exploit this insight. The reason for our design choice (i.e., to only iterate for AL rounds) in labeling is similar to selection: (1) too many and uncertain iteration rounds may introduce noise into the labeled data, and (2) iterating for AL rounds ensures that the highest abstraction level is reachable.

Example 4.3. Suppose the maximum abstraction level $AL = 2$ and the size of the set of abstraction points $|AS| = 3$. We use a natural number vector of length 3 to represent an abstraction for simplicity. The i-th element of the vector indicates the abstraction level of the i-th abstraction point. Initially, S_0 can be represented as 000. In the 1-st round of the labeling process, the search space is $\{000, 001, 010, 100, 011, 101, 110, 111\}$. BINGRAPH will choose an abstraction with minimum inversion count during the interaction. We assume it is $S_1 : 110$. Then, the feature values of the 1-st, and 2-nd abstraction points under abstraction S_0 will be labeled as 1, and the 3-rd abstraction point under abstraction S_0 will be labeled as 0. In the 2-nd round of the labeling process, the search space is $\{110, 111, 120, 210, 121, 211, 220, 221\}$. BINGRAPH will choose an abstraction with minimum inversion count during the interaction. We assume it is $S_2 : 211$. Then, the feature values of the 1-st, 3-rd abstraction points under abstraction S_1 will be labeled as 1, and the 2-nd abstraction point under abstraction S_1 will be labeled as 0.

The learning algorithm of BINGRAPH is inspired by GRAPHICK [Jeon et al. 2020]. The main idea is to find a range of real values such that it covers as many suitable labeled feature values and as few unsuitable labeled feature values as possible. Then, it is **highly likely to be beneficial** for the analysis when raising the abstraction level of an abstraction point whose feature value is contained in such a range. Since there are $\mathcal{N}$ types of property, we use $\mathcal{N}$-dimension cubes to represent a selection range. A $\mathcal{N}$-dimension cube $[l_1, r_1] \times [l_2, r_2] \times \cdots \times [l_{\mathcal{N}}, r_{\mathcal{N}}]$ includes all the feature values whose i-th element is contained in the segment $[l_i, r_i]$. The learning algorithm is shown in Algorithm 3, which generates a strategy *Learned*. *Learned* is a set of feature values, which **is implemented as a union of several $\mathcal{N}$-dimension cubes**. Formally, let $Cubes = \{[l_1, r_1] \times [l_2, r_2] \times \cdots \times [l_{\mathcal{N}}, r_{\mathcal{N}}] \mid l_i, r_i \in \mathbb{R}, l_i \leq r_i\}$ be the set of all $\mathcal{N}$-dimension cubes, then $Learned \in \{\bigcup_{C \in CubeSet} C \mid CubeSet \subseteq Cubes\} \subseteq \mathbb{R}^{\mathcal{N}}$. The learning algorithm maintains two sets of labeled feature values L_0 and L_1. They are points in $Labeled_0$ and $Labeled_1$ which have not yet been covered by *Learned*. The algorithm expands *Learned* through several rounds of computation until L_1 is empty. In each round, a $\mathcal{N}$-dimension cube $C \subseteq \mathbb{R}^{\mathcal{N}}$ is calculated by the procedure FINDCUBE with the remaining labeled sets of feature values L_0 and L_1. C will be contained in the final strategy *Learned* and the feature values contained by C will be remove from L_0 and L_1.

The algorithm of finding cubes is shown in Algorithm 4, which generates a $\mathcal{N}$-dimension cube. The output cube has proper coverage on remaining labeled sets of feature values L_0 and L_1. It starts with the cube containing each feature value in L_1 and gradually divides the cube into two smaller

Algorithm 4 Finding cubes during learning.

Input: Remained labeled sets of feature values $L_0, L_1 \subseteq \mathbb{R}^{\mathcal{N}}$.
Output: A $\mathcal{N}$-dimension cube $C \subseteq \mathbb{R}^{\mathcal{N}}$.

procedure FindCube(F_0, F_1)
  $m_i \leftarrow \min\{f_i \mid f \in L_1\}, M_i \leftarrow \max\{f_i \mid f \in L_1\}$
  $C \leftarrow [m_1, M_1] \times [m_2, M_2] \times \cdots \times [m_{\mathcal{N}}, M_{\mathcal{N}}]$
  repeat
    Random choose $i \in \{1, 2, \ldots, \mathcal{N}\}$
    Let $C = [l_1, r_1] \times [l_2, r_2] \times \cdots \times [l_{\mathcal{N}}, r_{\mathcal{N}}]$, $mid \leftarrow \frac{l_i + r_i}{2}$
    $C_{\mathrm{L}} \leftarrow [l_1, r_1] \times [l_2, r_2] \times \cdots \times [l_i, mid] \times \cdots \times [l_{\mathcal{N}}, r_{\mathcal{N}}]$
    $C_{\mathrm{R}} \leftarrow [l_1, r_1] \times [l_2, r_2] \times \cdots \times [mid, r_i] \times \cdots \times [l_{\mathcal{N}}, r_{\mathcal{N}}]$
    $C \leftarrow \arg\max_{C' \in \{C_{\mathrm{L}}, C_{\mathrm{R}}\}} \textsc{FeatureScore}(L_0, L_1, C')$
  until $\textsc{FeatureScore}(L_0, L_1, C) \geq \theta$ or *timeout*
  return C

cubes and picks the one with the higher FeatureScore. The definition of FeatureScore is:

$$\textsc{FeatureScore}(C, L_0, L_1) = \frac{|C \cap L_1|}{|C \cap (L_0 \cup L_1)|}$$

FeatureScore describes the fraction of feature values in L_1 in the cube. The division will terminate if the FeatureScore is not less than a hyper-parameter θ or it exceeds the time limit. The final cube will be returned.

Example 4.4. Consider $\mathcal{N} = 3$, $Labeled_0 = \{(3,3,3), (4,4,4), (5,5,5)\}$, $Labeled_1 = \{(1,1,1), (2,2,2), (4,4,4), (6,6,6)\}$ and the hyper-parameter $\theta = 0.6$. Initially, we set $L_0 \leftarrow Labeled_0$ and $L_1 \leftarrow Labeled_1$. The first cube starts from the smallest cube $[1,6] \times [1,6] \times [1,6]$ containing each feature value in L_1. The process of finding a cube is shown as follows, with the index randomly chosen:

$$\boxed{\begin{array}{c}[1,6] \times [1,6] \times [1,6] \\ \textsc{FeatureScore} = 0.57\end{array}} \xrightarrow{\text{choose } i=1} \boxed{\begin{array}{c}[1,3.5] \times [1,6] \times [1,6] \\ \textsc{FeatureScore} = 0.67\end{array}}$$

Then, the first cube $[1,3.5] \times [1,6] \times [1,6]$ is found. Feature values contained by this cube are removed. Now we have $L_0 = \{(4,4,4), (5,5,5)\}$ and $L_1 = \{(4,4,4), (6,6,6)\}$. The second cube starts from the smallest cube $[4,6] \times [4,6] \times [4,6]$ containing each feature value in L_1. The process of finding a cube is shown following:

$$\boxed{\begin{array}{c}[4,6] \times [4,6] \times [4,6] \\ \textsc{FeatureScore} = 0.5\end{array}} \xrightarrow{\text{choose } i=2} \boxed{\begin{array}{c}[4,6] \times [5,6] \times [4,6] \\ \textsc{FeatureScore} = 0.5\end{array}} \xrightarrow{\text{choose } i=2} \boxed{\begin{array}{c}[4,6] \times [5.5,6] \times [4,6] \\ \textsc{FeatureScore} = 1\end{array}}$$

Finally, the second cube $[4,6] \times [5.5,6] \times [4,6]$ is found. Feature values contained by this cube are removed. Now we have $L_0 = \{(4,4,4), (5,5,5)\}$ and $L_1 = \emptyset$. Therefore, the learning process is terminated and the result is $Learned = [1,3.5] \times [1,6] \times [1,6] \cup [4,6] \times [5.5,6] \times [4,6]$.

5 EXPERIMENTAL EVALUATION

Our evaluation aims to answer the following questions:

RQ1. How effective is BinGraph at optimization for generalization ability of Bayesian program analysis?

RQ2. How sensitive is BinGraph to the hyper-parameter and training benchmarks?

Table 4. Statistics of the instance analyses.

Analysis	# Input relations	# Output relations	# Derivation rules
Datarace analysis	58	44	102
Thread-escape analysis	34	27	60

RQ3. Is it necessary to calculate the derivation graph after an overall refinement to characterize abstraction points?

RQ4. How scalable is Bayesian program analysis using the abstraction selected by BinGraph?

RQ5. Can an existing abstraction selection approach for conventional program analyses replace our approach? Does an abstraction with a good balance of precision/scalability in a conventional analysis happen to be one with good generalization in its Bayesian counterpart?

We describe our experimental setup in Section 5.1, then discuss answers to the above questions in Section 5.2 to Section 5.6.

5.1 Experimental Setup

We conducted all experiments on Linux machines with 2.6 GHz processors and 256 GB RAM running Oracle HotSpot JVM 1.6. We use the Chord framework [Naik 2006] for Datalog program analysis and the Bingo framework [Raghothaman et al. 2018] for Bayesian inference. We set a size limit of 40 GB for the derivation graph and a time limit of 2 hours for one run of the inference on Bayesian networks. Exceeding one of these limits will be labeled as *failed* and be terminated.

Instance analyses. We summarize statistics of our two instance analyses in Table 4. (1) The first is a datarace analysis [Naik et al. 2006] that finds all possible statement pairs which may operate on the same heap object simultaneously with at least one write operation. It includes a parametric flow-insensitive and context-sensitive k-object-sensitive pointer analysis [Milanova et al. 2005]. The abstraction points are the allocation sites in the k-object-sensitive analysis, and the abstraction level (i.e., the k value) for each allocation site is in $\{0, 1, 2, 3\}$. The abstraction levels indicate the degree of context-sensitivity and the site is handled in a context-insensitive way when the corresponding level is 0. (2) To demonstrate the generality of BinGraph, we also consider a thread-escape analysis [Naik et al. 2012] that finds all possible statements whose operation target may be accessed by multiple threads. The analysis is flow- and context-insensitive. The analysis is parameterized by how the heap objects are modeled. The abstraction points are also allocation sites and the abstraction level for each allocation site is in $\{0, 1\}$. All objects in allocation sites of level 0 will be considered as one object together during analysis. For an allocation site of level 1, a standalone abstract object is created and all objects created at the site will be considered as it.

Benchmarks. We evaluated BinGraph on 13 benchmarks shown in Table 5, including programs from the DaCapo suite [Blackburn et al. 2006] and from past works. For each benchmark, we previously check each alarm whether it is true and simulate the whole interaction automatically to calculate relevant metrics. For the datarace analysis, we obtain true alarms by manual inspection [Raghothaman et al. 2018]. For the thread-escape analysis, we use the result of a CEGAR-based flow- and context-sensitive analysis [Zhang et al. 2013] as true alarms.[§] Four programs are excluded from the datarace evaluation since the analysis generates no alarms for these programs. Another

[§]Since manual inspection of real bugs requires a lot of manual effort, it is common practice in program analysis studies [Jeon et al. 2020; Li et al. 2018a; Mangal et al. 2015; Zhang et al. 2017] to use an accurate but heavy analysis to obtain ground truth for interaction or comparison. In addition to this, our approach can be viewed as a lightweight but effective way to approximate an accurate but heavy analysis [Zhang et al. 2013].

Table 5. Benchmark characteristics. "Total" and "App" are numbers with and without the JDK using 0-CFA call graph construction. "DA" and "TEA" are the datarace analysis and the thread-escape analysis. "-" denotes the benchmark is not used in the analysis.

Program	Description	# Classes		# Methods		Bytecode (KB)		Source (KLOC)		# True alarms	
		App	Total	App	Total	App	Total	App	Total	DA	TEA
ftp	Apache FTP server	119	1,196	608	7,650	35	443	17	305	75	643
javasrc-p	Java source code to HTML translator	51	1,009	471	6,624	42	403	12	276	-	695
jspider	Web spider engine	113	1,193	426	7,431	17	429	6.7	298	9	430
hedc	Web crawler from ETH	44	1,157	230	7,501	15	464	6	292	12	287
montecarlo	Financial simulator	18	974	115	6,260	5	365	3.5	266	-	54
pool	Apache Commons Pool	27	1,132	194	7,313	7.5	417	5.3	302	-	312
raytracer	3D raytracer	18	105	74	391	4.9	23	1.8	55	3	233
toba-s	Java bytecode to C compiler	25	985	154	6,338	31	393	6.2	270	-	998
weblech	Website download/mirror tool	56	1,276	303	8,421	18	503	10	322	6	276
avrora	AVR microcontroller simulator	1,119	2,080	3,875	10,095	191	553	54	318	29	-
luindex	Document indexing tool	169	1,164	1,030	7,461	72	453	30	299	2	-
sunflow	Photo-realistic image rendering system	127	1,853	967	12,901	87	878	15	529	171	-
xalan	XML to HTML transforming tool	390	1,723	3,007	12,181	159	786	119	495	75	-

Table 6. The type of properties used in experiments with $\mathcal{N} = 12$.

Number	Type	Number	Type
1	The count of reachable vertices	7	The count of vertices with shortest distance ≤ 5
2	Average of shortest distance to reachable vertices	8	The count of vertices with shortest distance ≤ 6
3	The count of vertices with shortest distance ≤ 1	9	The count of vertices with shortest distance ≤ 7
4	The count of vertices with shortest distance ≤ 2	10	The count of vertices with shortest distance ≤ 8
5	The count of vertices with shortest distance ≤ 3	11	The count of vertices with shortest distance ≤ 9
6	The count of vertices with shortest distance ≤ 4	12	The count of vertices with shortest distance ≤ 10

four programs are excluded from the thread-escape analysis evaluation because the oracle analysis fails to terminate on them.

Baseline abstractions. We compare abstractions produced by BinGraph to three baseline abstractions, Base-C, Base-P and Base-R. Base-C corresponds to the coarsest abstraction $\mathcal{S}_0$. Base-P corresponds to the most precise abstraction where $S_{\mathrm{P}}(x) = AL$ for $x \in AS$. Base-R corresponds to the random abstraction $S_{\mathrm{R}}(x)$ where abstraction levels are uniformly distributed in $\{0, 1, \ldots, AL\}$. For Base-R, we show the average measurement across 3 runs. We compare BinGraph to these baselines to demonstrate that the direction in which BinGraph makes an abstraction more precise is beneficial to the generalization ability, being neither too coarse nor too fine, and not degenerating to a random selection. We will evaluate abstractions with a good balance between precision/scalability in conventional program analysis in Section 5.6.

Features. We present the type (i.e., the meaning of $\beta_i(G, v)$) and number (i.e., the value of $\mathcal{N}$) of properties used of BinGraph in Table 6. These properties characterize abstraction points straightforwardly. We recommend using BinGraph with the properties used in the experiments since they show good optimization results. To avoid dividing by 0, all properties will be added by 1 during calculating feature values.

Metrics. The main metric is the inversion count introduced in Section 3.3, which reflects the user experience during the overall interaction and further reflects the generalization ability of a Bayesian program analysis. We also demonstrate two metrics used in previous research [Raghothaman et al. 2018] for supplement: Rank-100%-T represents the rounds for inspecting all true alarms by the user. Rank-90%-T represents that for inspecting 90% true alarms (rounding up) by the user.

Table 7. Summary of metrics for effectiveness of BinGraph. "Average" represents the average reduction ratio compared to baselines.

	Program	Inversion				Rank-100%-T				Rank-90%-T			
		BinGraph	Base-C	Base-P	Base-R	BinGraph	Base-C	Base-P	Base-R	BinGraph	Base-C	Base-P	Base-R
Datarace analysis	avrora	6,249	3,852	6,944	7,341	761	938	717	744	364	421	392	415
	ftp	388	1,173	432	540	84	169	83	86	77	112	75	77
	sunflow	10,055	17,790	*failed*	14,443	359	460	*failed*	961	254	429	*failed*	315
	raytracer	19	87	18	37	10	32	9	15	10	32	9	15
	luindex	22	32	646	236	13	18	325	120	13	18	325	120
	xalan	*failed*	14,868	*failed*	*failed*	*failed*	326	*failed*	*failed*	*failed*	319	*failed*	*failed*
	Average		31.52%↓	27.81%↓	42.55%↓		37.53%↓	19.39%↓	37.04%↓		36.42%↓	22.34%↓	30.83%↓
Thread-escape analysis	hedc	5,991	14,381	7,045	10,065	372	396	379	393	303	337	316	325
	jspider	20,033	36,838	35,517	40,411	635	662	618	629	451	500	527	540
	montecarlo	496	1,812	583	1,969	78	163	78	154	59	156	61	122
	pool	9,591	20,226	10,450	18,423	371	402	395	411	323	369	361	377
	raytracer	6,101	9,756	6,822	8,353	287	314	319	339	263	287	286	281
	toba-s	53,419	118,509	76,604	104,175	1,278	1,222	1,278	1,256	960	1,052	1,149	1,103
	Average		53.59%↓	20.42%↓	48.22%↓		12.34%↓	2.53%↓	12.84%↓		18.61%↓	9.47%↓	18.10%↓

Learning configuration. We divide benchmarks into training/validation/test sets. The validation set is used to determine the hyper-parameter θ introduced in Section 4.2. For the datarace analysis, we use {`jspider`, `hedc`} for training and {`weblech`} for validation. For the thread-escape analysis, we use {`ftp`, `javasrc-p`} for training and {`weblech`} for validation. We choose these benchmarks as they are among the smaller benchmarks in the size of bytecode since the time cost of using large programs is unacceptable for training and validation. Other benchmarks are used for test.

5.2 Effectiveness

We present the summary of the metrics for the effectiveness result in Table 7. Compared to the baselines, BinGraph has significantly fewer inversion counts on most benchmarks. BinGraph outperforms Base-C on 10 of 12 benchmarks, with an average reduction ratio of 31.52% and 53.59% for the two analyses respectively. This shows that BinGraph improves generalization ability by making abstractions more precise. BinGraph outperforms Base-P on 11 of 12 benchmarks, with an average reduction ratio of 27.81% and 20.42% for the two analyses respectively. This shows that the abstractions selected by BinGraph are not as precise as Base-P, but have much better generalization ability. This is because BinGraph only raises the levels of abstraction points that are beneficial to generalization. A typical example is `luindex`, in which Base-P has 29× inversion count compared to BinGraph because the links between false alarms are cut under high precision abstractions. BinGraph outperforms Base-R on 12 of 12 benchmarks, with an average reduction ratio of 42.55% and 48.22% for the two analyses. This shows that the strategy learned by BinGraph is quite different from random selections. Moreover, there are similar improvements in the other two metrics.

We plot ROC curves [Fawcett 2006] for the datarace analysis in Figure 8 and for the thread-escape analysis in Figure 9. A point (x, y) represents that the user has inspected x false alarms and y true alarms after $(x + y)$ rounds of interaction. A relevant metric AUC is the normalized area under the ROC curve, which is used in previous research [Raghothaman et al. 2018]. The relation between inversion counts and AUC is $\textsc{Inversion}(S) = N_T N_F(1 - \text{AUC})$, where N_T and N_F are numbers of true alarms and false alarms. Since the number of false alarms may differ due to abstractions, we use inversion counts instead of AUC as the major metric. The larger the AUC is, the lower the inversion count is. Therefore, AUC can visually show the difference in generalization ability

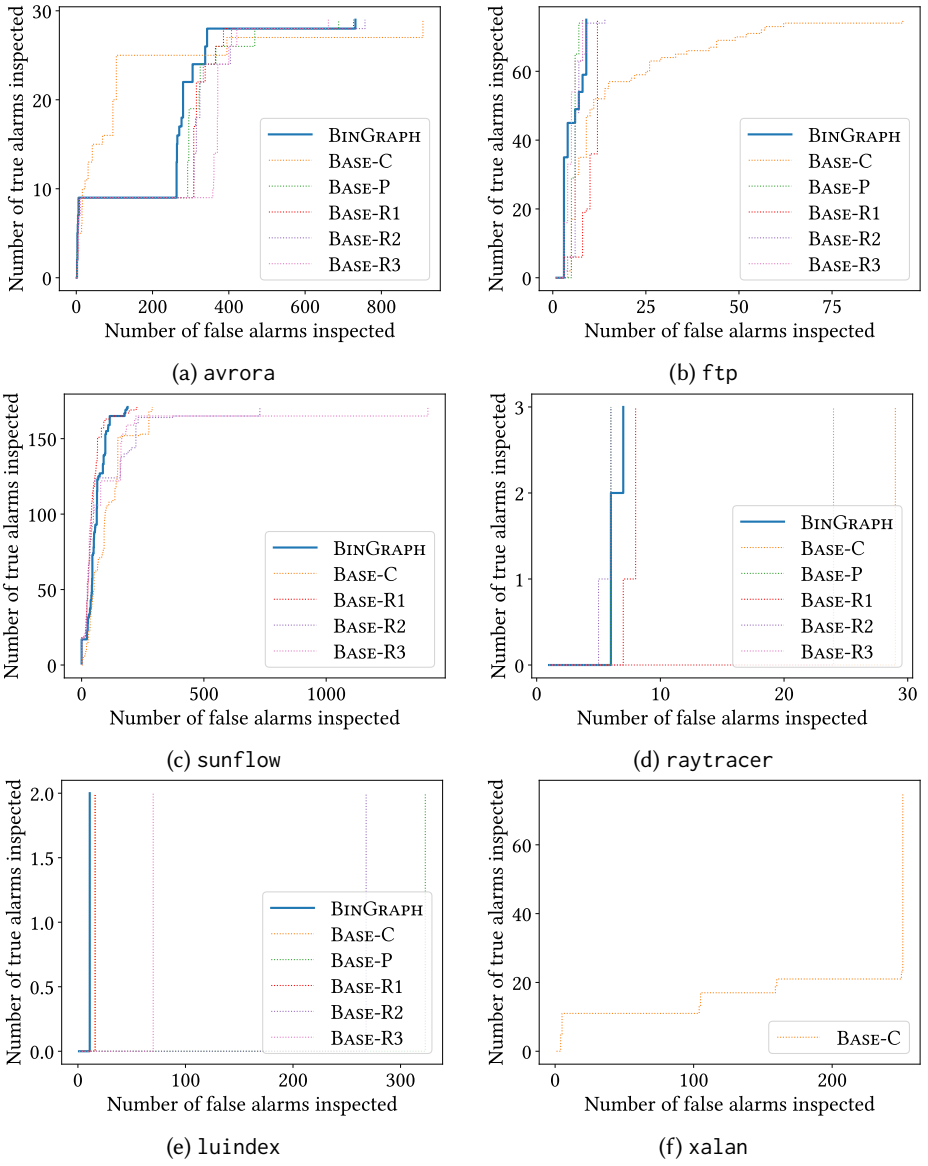


Fig. 8. The ROC curves for the datarace analysis. Base-R1 to Base-R3 correspond to 3 runs of Base-R. Results for *failed* configurations are not displayed.

between different abstractions. Compared to other baselines, BinGraph can be clearly seen to have significantly higher AUC. This is a visual illustration of the powerful generalization ability of abstractions BinGraph selects.

There are two outliers. One is that Base-C has a lower inversion count than other approaches for benchmark `avrora` in the datarace analysis experiment, but has higher rank-100%-T and rank-90%-T. The main reason is that Base-C finds 25 true alarms in only 130 rounds (while BinGraph, Base-P and Base-R use 363, 391, and 414 rounds, respectively), but takes 808 rounds to find the remaining 4

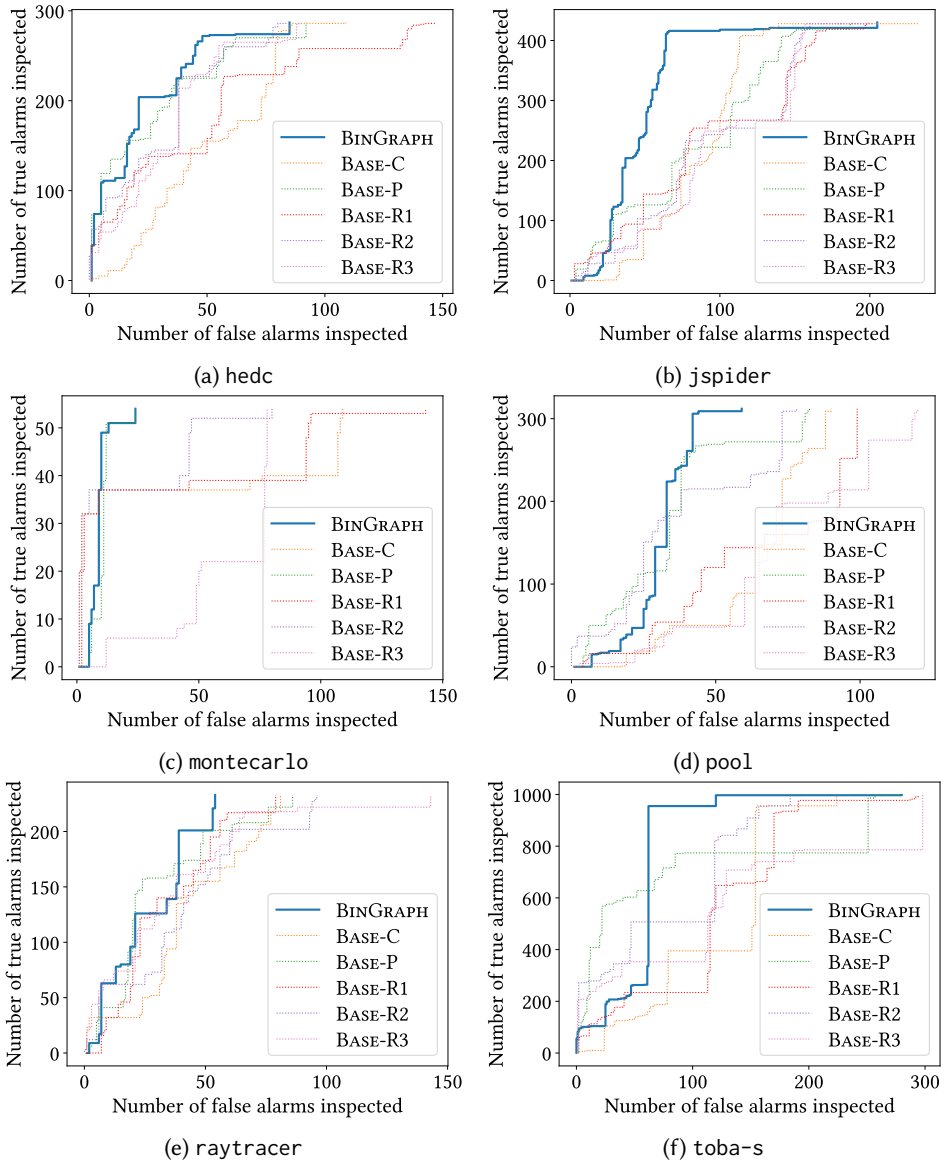
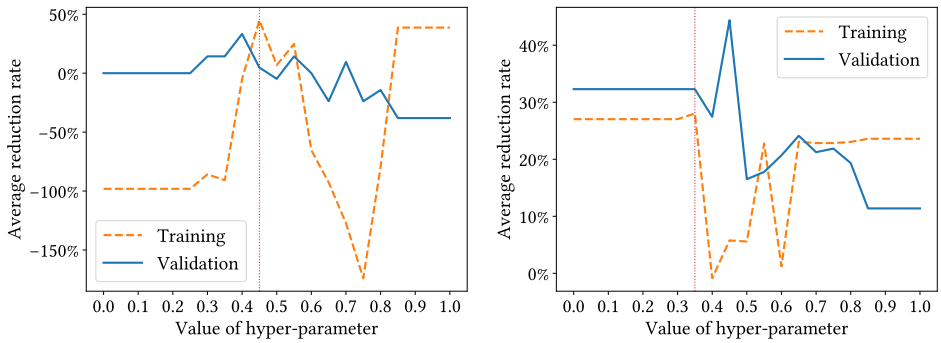


Fig. 9. The ROC curves for the thread-escape analysis. Base-R1 to Base-R3 correspond to 3 runs of Base-R.

true alarms. This shows that the performance of Base-C is good at the beginning of the interaction, but is very bad afterward. For this particular reason, Base-C has a very low inversion count, but this does not mean that Base-C has a stronger generalization ability. Another exception is that most valuable abstractions, except the coarsest one 0-CFA, are labeled as *failed* for benchmark `xalan` in the datarace analysis, including 1-object-sensitivity and 2-object-sensitivity (i.e., $S(x) = 1$ or 2 respectively holds for $x \in AS$ in the abstraction). The main reason is that existing Bayesian program analysis frameworks are not scalable for large programs using precise abstractions.



(a) Datarace analysis (b) Thread-escape analysis

Fig. 10. Performance of BinGraph on training and validation sets with different hyper-parameter θ. The vertical dotted lines represent the chosen θ.

Table 8. Summary of metrics for the leave-one-out cross-validation in the thread-escape analysis. Statistics are average reduction compared to Base-R.

Setting	Inversion	Rank-100%-T	Rank-90%-T
BinGraph	48.22%↓	12.84%↓	18.10%↓
BinGraph$_C$	40.76%↓	11.34%↓	15.69%↓

In summary, BinGraph is indeed effective at optimizing for the generalization ability of Bayesian program analysis, and can potentially improve user experience during interaction.

5.3 Sensitivity Study

We evaluate the sensitivity of BinGraph to hyper-parameter and training benchmarks.

Sensitivity to the hyper-parameter. We present the performance of BinGraph on training and validation sets with different hyper-parameter θ values (see Algorithm 4) in Figure 10. We iterate θ from 0 to 1 with 0.05 intervals and evaluate the effect of it using the average reduction ratio of inversion count compared to Base-R. The performance of BinGraph varies with the change of θ. For example, when θ is close to 1, its performance is significantly better on the training set than that on the validation set. The main reason is that Algorithm 4 will generate low coverage cubes when θ is high, which will cause over-fitting. To counter the sensitivity of BinGraph to hyper-parameter, we choose θ with the highest combined average reduction ratio among training and validation sets. The chosen θ values are 0.35 and 0.45 for the thread-escape analysis and the datarace analysis, respectively.

Sensitivity to training benchmarks. In order to measure the sensitivity of BinGraph to training benchmarks, we conducted leave-one-out cross-validation. Since some benchmarks for the datarace analysis are too large to be used for training, we only studied the thread-escape analysis with all 9 benchmarks. Let BinGraph$_C$ be the setting of cross-validation, we present the summary of metrics in Table 8. It can be shown that BinGraph and BinGraph$_C$ have a similar improvement over Base-R, with the differences only being 7.46%, 1.50%, and 2.41% in the three metrics. In summary, BinGraph is not sensitive to the selection of training benchmarks.

Table 9. Summary of metrics for the ablation experiment. Statistics are average reduction (or increase) compared to Base-R.

Setting	Datarace analysis			Thread-escape analysis		
	Inversion	Rank-100%-T	Rank-90%-T	Inversion	Rank-100%-T	Rank-90%-T
BinGraph	42.55%↓	37.04%↓	30.83%↓	48.22%↓	12.84%↓	18.10%↓
BinGraph$_A$	2.54%↓	1.49%↓	4.48%↑	28.45%↑	13.40%↑	10.00%↑

5.4 Necessity to Use Two Derivation Graphs

Note that BinGraph uses two derivation graphs to characterize each abstraction point. However, the computational cost on the derivation graph after an overall refinement G' is significantly higher than that on the derivation graph before an overall refinement G. To validate the necessity to calculate G', we conduct an ablation experiment that only uses G to characterize each abstraction point. Formally, we redefine the feature value in Section 4.1 as $\lambda_i(S, x) = \beta_i(\textsc{Graph}(S), AI(x, S(x))$.

For instance, the feature value of `H1` in Table 3 will become $(35, 4.114, 18)$. Other settings remain unchanged, including the training set, validation set, and the criteria to choose hyper-parameter θ. Let BinGraph$_A$ be the setting of the ablation experiment, we present the summary of metrics in Table 9. It can be shown that the variation of BinGraph$_A$ compared to Base-R is completely different from BinGraph, and even worse than Base-R in most metrics. The main reason is that G only characterizes each abstraction point based on the information in the current abstraction but not that after potential refinements, which leads to the learned strategy being not beneficial in finding the suitable abstraction. Overall, using G' is necessary and valuable.

5.5 Scalability

We evaluate the scalability of the two parts of BinGraph separately. In the offline part, we evaluate the training cost of BinGraph. In the online part, we evaluate the running cost of the Bayesian program analysis under the abstraction selected by BinGraph. Since the running cost of the Datalog engine is negligible for the overall running process (less than 1%), we only present the evaluation of Bayesian inference.

Training cost. To speed up the training process, we implemented parallelism in (1) Simulated Annealing for labeling, and (2) validation for obtaining the hyper-parameter. For the datarace analysis, the labeling process took 4 days and the learning process (including validation) took 1 day. For the thread-escape analysis, the labeling process took 2 days and the learning process (including validation) took 8 hours. In total, the training cost of BinGraph is acceptable.

Bayesian inference cost. The computational cost of Bayesian inference depends mainly on the size of the derivation graph after reduction. Since the average iteration times for the thread-escape analysis are negligible (less than 1 second), we do not present them. We present the summary of metrics for the scalability of the datarace analysis in Table 10. It is counter-intuitive that the most precise abstraction Base-P has a smaller derivation graph than other abstractions in some benchmarks such as `ftp`. However, the derivation graph before the reduction of Base-P is the biggest one. The main reason is that the algorithm to remove cycles is heuristic, and its effect depends on the structure of the graph. The conclusion is that, under the existing framework of Bayesian program analysis, the scalability of BinGraph is acceptable compared to other approaches.

Table 10. Summary of metrics for the Bayesian inference cost of BinGraph in the datarace analysis. Tuples and relevant ground clauses are counted from the derivation graph after reduction. Iteration times are average values during the interaction.

Program	# Tuples				# Relevant ground clauses				Iteration time (s)			
	BinGraph	Base-C	Base-P	Base-R	BinGraph	Base-C	Base-P	Base-R	BinGraph	Base-C	Base-P	Base-R
avrora	44,613	84,507	59,061	47,971	33,467	85,585	48,285	36,899	97	612	189	114
ftp	65,931	115,621	19,269	27,905	63,940	116,628	13,593	21,474	309	680	25	86
sunflow	84,561	390,638	*failed*	76,065	73,711	420,254	*failed*	62,069	456	3,044	*failed*	415
raytracer	7,121	4,750	6,886	6,383	5,480	3,175	5,342	4,756	5.9	4.0	6.9	6.3
luindex	36,941	53,154	32,167	43,992	24,261	42,210	20,625	32,118	22	198	27	134

Table 11. Summary of metrics for the experiments on conventional approaches. Statistics are average reduction (or increase) compared to Base-R.

Setting	Datarace analysis			Thread-escape analysis		
	Inversion	Rank-100%-T	Rank-90%-T	Inversion	Rank-100%-T	Rank-90%-T
BinGraph	42.55%↓	37.04%↓	30.83%↓	48.22%↓	12.84%↓	18.10%↓
$\text{BinGraph}_{\text{M}}$	267.16%↑	27.42%↑	31.55%↑	40.49%↓	16.99%↓	17.72%↓
Minimal	-	-	-	27.29%↓	15.12%↓	13.00%↓

5.6 Ineffectiveness of Conventional Approaches

Conventional approaches aim to find abstractions with a good balance between precision and scalability. Are these abstractions also good for generalization and can we use a conventional approach to replace our approach? To answer these questions, we first conducted a controlled variable experiment where we modify BinGraph such that it learns such abstractions. Given a training program, we apply BinGraph to learn an abstraction that is the cheapest abstraction among all the abstractions producing the least alarms. We refer to these abstractions as *minimal abstractions* and find them using a systematic search LearnMinimalAbstraction [Jeon et al. 2020]. Further, since our approach is not originally designed for learning these abstractions, to remove the noise incurred by learning, we apply the systematic search to identify minimal abstractions on the test programs and evaluate their generalization effect. We only study the results of the thread-escape analysis because the systematic search does not scale for the datarace analysis when using the test programs. Table 11 shows how much these abstractions identified under the two settings improve over the cheapest abstraction when applied to a Bayesian program analysis. $\text{BinGraph}_{\text{M}}$ and Minimal denote the results under the two setting respectively.

The result shows that $\text{BinGraph}_{\text{M}}$ has a similarly good performance compared to our approach on the thread-escape analysis, but has extremely bad performance on the datarace analysis. The reason is that for the thread-escape dataset, some of the abstractions that balance precision and scalability happen to be abstractions with good generalization ability. But this is not the case for the datarace analysis. Moreover, the performance of Minimal is significantly worse than BinGraph and $\text{BinGraph}_{\text{M}}$. This shows that the minimal abstractions on the test programs for the thread-escape analysis are actually worse in terms of generalization. As a result, conventional approaches cannot reliably find abstractions with good generalization ability for Bayesian program analyses.

6 RELATED WORK

Our approach is related to research on Bayesian program analysis and data-driven abstraction selection techniques for conventional analysis. We summarize the related prior works below.

Bayesian program analysis. Bayesian program analysis tools build probabilistic models based on logical rules, generalize various posterior information, and calculate the probability for each alarm to be true. Eugene [Mangal et al. 2015] and Bingo [Raghothaman et al. 2018] use user feedback as posterior information. Drake [Heo et al. 2019b] uses information from programs of old versions and DynaBoost [Chen et al. 2021] uses dynamic analysis results. Since using different abstractions is equivalent to using different logical rules and does not cause incompatibility in subsequent procedures (such as building probabilistic models and generalizing posterior information), our approach can be directly combined with these tools. BayeSmith [Kim et al. 2022] learns new derivation rules and probabilities from existing rules using syntactic information. For an existing rule R_1 : A(x) :- B(x), BayeSmith may refine it to two rules R_{11} : A(x) :- B(x), Loop(x) and R_{12} : A(x) :- B(x), ¬Loop(x) with different probabilities. For each ground clause of R_1, it will be replaced by one of R_{11} or one of R_{12}. The only difference in the generated Bayesian network is new input tuples Loop(x), ¬Loop(x), and new probabilities of the ground clauses. Since input tuples have probabilities of 1, they do not affect Bayesian inference. Therefore, BayeSmith does not substantially change the structure of Bayesian networks, and it can be equated to an approach to learning probabilities of ground clauses. Instead, our approach changes the structure of Bayesian networks rather than probabilities of ground clauses, so that our approach and BayeSmith are actually complementary to each other.

Data-driven abstraction selection for conventional analysis. The abstraction selection problem for conventional program analysis has been extensively studied [Bielik et al. 2017; Grigore and Yang 2016; He et al. 2020; Heo et al. 2016, 2019a, 2017; Jeon et al. 2019, 2018, 2020; Jeon and Oh 2022; Jeong et al. 2017; Liang et al. 2011; Oh et al. 2015; Peleg et al. 2016; Singh et al. 2018; Wei and Ryder 2015]. Although these approaches are not effective in optimizing for generalization, some of the ideas are worth learning from. Our approach adapts some of them [Jeon et al. 2019, 2018, 2020; Jeon and Oh 2022; Jeong et al. 2017; Oh et al. 2015], which use features to express the characteristics of each abstraction point independently. In these approaches, the chosen features are related to the program itself (such as whether a method has an allocation site or not), or the analysis itself (such as the degree of a vertex in the object allocation graph [Li et al. 2018b; Tan et al. 2016]). For generality and effectiveness in the context of Bayesian program analysis, our approach uses properties based on differences in the derivation graphs before and after an overall refinement.

7 CONCLUSION

We present BinGraph, a general framework for learning abstraction selection for Bayesian program analysis. The main idea of BinGraph is refining the abstraction for several rounds and leveraging the difference of derivation graphs to characterize each abstraction point. In the experiments with two instance analyses and 13 Java programs, we demonstrate the effectiveness of BinGraph in enhancing the generalization ability of Bayesian program analysis.

REFERENCES

Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2017. Learning a Static Analyzer from Data. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10426)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 233–253. https://doi.org/10.1007/978-3-319-63387-9_12

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, Peri L. Tarr and William R. Cook (Eds.). ACM, 169–190. https://doi.org/10.1145/1167473.1167488

Tianyi Chen, Kihong Heo, and Mukund Raghothaman. 2021. Boosting static analysis accuracy with instrumented test executions. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 1154–1165. https://doi.org/10.1145/3468264.3468626

Patrick Cousot. 1996. Abstract Interpretation. *ACM Comput. Surv.* 28, 2 (1996), 324–328. https://doi.org/10.1145/234528.234740

Tom Fawcett. 2006. An introduction to ROC analysis. *Pattern Recognit. Lett.* 27, 8 (2006), 861–874. https://doi.org/10.1016/j.patrec.2005.10.010

Radu Grigore and Hongseok Yang. 2016. Abstraction refinement guided by a learnt probabilistic model. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 485–498. https://doi.org/10.1145/2837614.2837663

Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017, Barcelona, Spain, June 18, 2017*, Karim Ali and Cristina Cifuentes (Eds.). ACM, 13–18. https://doi.org/10.1145/3088515.3088519

Jingxuan He, Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2020. Learning fast and precise numerical analysis. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1112–1127. https://doi.org/10.1145/3385412.3386016

Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2016. Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9837)*, Xavier Rival (Ed.). Springer, 237–256. https://doi.org/10.1007/978-3-662-53413-7_12

Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2019a. Resource-aware program analysis via online abstraction coarsening. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 94–104. https://doi.org/10.1109/ICSE.2019.00027

Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-learning-guided selectively unsound static analysis. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 519–529. https://doi.org/10.1109/ICSE.2017.54

Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik. 2019b. Continuously reasoning about programs using differential Bayesian inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 561–575. https://doi.org/10.1145/3314221.3314616

Minseok Jeon, Sehun Jeong, Sung Deok Cha, and Hakjoo Oh. 2019. A Machine-Learning Algorithm with Disjunctive Model for Data-Driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 41, 2 (2019), 13:1–13:41. https://doi.org/10.1145/3293607

Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and scalable points-to analysis via data-driven context tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 140:1–140:29. https://doi.org/10.1145/3276510

Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 179:1–179:30. https://doi.org/10.1145/3428247

Minseok Jeon and Hakjoo Oh. 2022. Return of CFA: call-site sensitivity can be superior to object sensitivity even for object-oriented programs. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498720

Sehun Jeong, Minseok Jeon, Sung Deok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 100:1–100:28. https://doi.org/10.1145/3133924

George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 423–434. https://doi.org/10.1145/2491956.2462191

Hyunsu Kim, Mukund Raghothaman, and Kihong Heo. 2022. Learning Probabilistic Models for Static Analysis Alarms. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1282–1293. https://doi.org/10.1145/3510003.3510098

Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi. 1983. Optimization by Simmulated Annealing. *Sci.* 220, 4598 (1983), 671–680. https://doi.org/10.1126/science.220.4598.671

Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models - Principles and Techniques*. MIT Press. https://dl.acm.org/doi/10.5555/1795555

Haofeng Li, Jie Lu, Haining Meng, Liqing Cao, Yongheng Huang, Lian Li, and Lin Gao. 2022. Generic sensitivity: customizing context-sensitive pointer analysis for generics. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 1110–1121. https://doi.org/10.1145/

3540250.3549122

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018a. Precision-guided context sensitivity for pointer analysis. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 141:1–141:29. https://doi.org/10.1145/3276511

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018b. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 129–140. https://doi.org/10.1145/3236024.3236041

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 42, 2 (2020), 10:1–10:40. https://doi.org/10.1145/3381915

Percy Liang and Mayur Naik. 2011. Scaling abstraction refinement via pruning. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 590–601. https://doi.org/10.1145/1993498.1993567

Percy Liang, Omer Tripp, and Mayur Naik. 2011. Learning minimal abstractions. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 31–42. https://doi.org/10.1145/1926385.1926391

Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 148:1–148:29. https://doi.org/10.1145/3360574

Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. 2015. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 462–473. https://doi.org/10.1145/2786805.2786851

Ana L. Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (2005), 1–41. https://doi.org/10.1145/1044834.1044835

Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. 1999. Loopy Belief Propagation for Approximate Inference: An Empirical Study. In *UAI '99: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence, Stockholm, Sweden, July 30 - August 1, 1999*, Kathryn B. Laskey and Henri Prade (Eds.). Morgan Kaufmann, 467–475. https://dl.acm.org/doi/10.5555/2073796.2073849

Mayur Naik. 2006. *Chord: A program analysis platform for Java.* https://bitbucket.org/psl-lab/jchord/src/master/

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 308–319. https://doi.org/10.1145/1133981.1134018

Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. 2012. Abstractions from tests. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 373–386. https://doi.org/10.1145/2103656.2103701

Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective context-sensitivity guided by impact pre-analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 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, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 572–588. https://doi.org/10.1145/2814270.2814309

Hila Peleg, Sharon Shoham, and Eran Yahav. 2016. Dˆ3 : Data-Driven Disjunctive Abstraction. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 185–205. https://doi.org/10.1007/978-3-662-49122-5_9

Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided program reasoning using Bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 722–735. https://doi.org/10.1145/3192366.3192417

Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2018. Fast Numerical Program Analysis with Reinforcement Learning. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 211–229. https://doi.org/10.1007/978-3-319-96145-3_12

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 485–495. https://doi.org/10.1145/

2594291.2594320

Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. https://doi.org/10.1145/3485524

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 (Lecture Notes in Computer Science, Vol. 9837)*, Xavier Rival (Ed.). Springer, 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, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 278–291. https://doi.org/10.1145/3062341.3062360

Shiyi Wei and Barbara G. Ryder. 2015. Adaptive Context-sensitive Analysis for JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs, Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 712–734. https://doi.org/10.4230/LIPIcs.ECOOP.2015.712

Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. 2017. Effective interactive resolution of static analysis alarms. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 57:1–57:30. https://doi.org/10.1145/3133881

Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in Datalog. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 239–248. https://doi.org/10.1145/2594291.2594327

Xin Zhang, Mayur Naik, and Hongseok Yang. 2013. Finding optimum abstractions in parametric dataflow analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 365–376. https://doi.org/10.1145/2491956.2462185

Received 21-OCT-2023; accepted 2024-02-24