RESEARCH-ARTICLE

On Abstraction Refinement for Bayesian Program Analysis

YUANFENG SHI, Peking University, Beijing, China

YIFAN ZHANG, Peking University, Beijing, China

XIN ZHANG, Peking University, Beijing, China

Open Access Support provided by:

Peking University

Published: 09 October 2025
Accepted: 12 August 2025
Received: 26 March 2025

On Abstraction Refinement for Bayesian Program Analysis

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

Bayesian program analysis is a systematic approach to learn from external information for better accuracy by converting logical deduction in conventional program analysis into Bayesian inference. A key challenge in Bayesian program analysis is how to select program abstractions to effectively generalize from external information. A recent approach addresses this challenge by learning a selection policy on training programs but may result in sub-optimal performance on new programs due to its learning nature and when the training set selection is not ideal. To address this problem, we propose an approach that is inspired by the framework of counterexample-guided refinement to search for an abstraction on the fly. Our key innovation is to apply the theory of conditional independence to refine the abstraction so that incorrect generalizations can be removed. To demonstrate the effectiveness of our approach, we have instantiated it on a Bayesian thread-escape analysis and a Bayesian datarace analysis and shown that it significantly improves the performance of the analyses.

CCS Concepts: • **Software and its engineering → Automated static analysis**.

Additional Key Words and Phrases: Static Analysis, Bayesian Network, Alarm Ranking, Machine Learning for Program Analysis, Abstraction Refinement

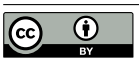
ACM Reference Format:
Yuanfeng Shi, Yifan Zhang, and Xin Zhang. 2025. On Abstraction Refinement for Bayesian Program Analysis. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 388 (October 2025), 27 pages. https://doi.org/10.1145/3763166

1 Introduction

Bayesian program analysis is a new type of program analysis that emerged in recent years [11, 19, 30, 39, 52, 53]. More than a set of alarms which a traditional analysis reports, it additionally computes the confidence value of each alarm. The main idea behind such analyses is integrating probability into the traditional logic-based program analysis [52]. Each analysis rule is assigned with a parameter as the probability to be valid. While the probability part can handle uncertainties in analysis design, the logic part retains the advantages of being concise, precise, explainable, and rigorously formal as before. Besides, by converting the logical deduction of traditional analyses into Bayesian inference, this approach acquires learning abilities and automatic adaptation capabilities from the posterior information, such as learning users' feedback and incorporating dynamic information. For instance, the framework named Bingo [39] can learn from users' binary feedback on certain alarms and

*Corresponding author.

Authors' Contact Information: 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; 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; 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.



ACM 2475-1421/2025/10-ART388
https://doi.org/10.1145/3763166

generalize this information to produce better probabilities for other alarms. This in turn improves the overall ranking of these alarms, therefore improving the analysis' precision[1].

Just like traditional program analysis, the choice of program abstraction is crucial in Bayesian program analysis. As any non-trivial program analysis problem is theoretically undecidable in the worst case [41], program analysis requires abstracting program semantics. Actually, in both traditional and Bayesian program analyses, every single abstract program state is used to represent multiple possible concrete program states, and all reachable abstract program states will be computed to find all reachable concrete program states. This approach approximates a hard program analysis problem into a tractable one, and the abstraction granularity controls the precision and the scalability of the analysis. A key design problem is how to choose an abstraction that balances the trade-off between precision and scalability for complex real-world problems.

However, compared to the case of traditional analyses, abstraction granularity also controls the ability to generalize from the posterior information in Bayesian analyses. Since a Bayesian program analysis consists of a learning process, it suffers from the classical over-generalization (or over-fitting) problem. Under a coarse program abstraction, due to the confusion of semantic information, it may assign false relevance between facts and is more likely to regard features in the posterior information as general properties. As a result, the analysis will not only exclude false positive alarms, but also mistakenly reduce the confidence values of many true positive alarms. On the other hand, using a too fine-grained abstraction will limit the generalization of posterior information. Moreover, such posterior information (e.g., user labels) is usually expensive to obtain. Hence, it is critical to choose the proper abstraction granularity for Bayesian program analyses.

A recent work [53], BinGraph, addresses this problem by training a model that predicts a good abstraction in terms of generalization for a given program. However, such an offline learning-based approach can lead to sub-optimal performance due to the approximation nature of machine learning techniques and when the new program is out of the distribution of the training programs.

In this paper, we complement the above approach by proposing an online abstraction-search approach that is inspired by the classical counterexample-guided refinement (CEGAR) technique [14]. Given a family of abstractions, similar to CEGAR, our approach starts from a given abstraction and iteratively refines it. The abstraction can be the coarsest one in the family or produced by the offline learning approach. Different from CEGAR, instead of trying to eliminate analysis derivations that lead to false alarms, our approach tries to eliminate derivations that lead to incorrect generalizations. Meanwhile, our approach also tries to keep derivations that lead to correct generalizations.

The existence of probabilities adds additional complexities to our problem as the conventional logic-based approach to eliminate analysis derivation does not work. To address this issue, we leverage the theory of conditional independence [4, Chapter 8.2] and build an information-transmission graph from the analysis derivation. To keep correct generalizations and remove incorrect ones, the refinement problem is cast as a minimum-cut problem while preserving connectivity between some nodes on the graph. The problem has shown to be a NP-hard problem [15]. To solve it, we propose an effective maximum satisfiability [35] encoding and applies an existing solver [32].

We have implemented our approach in a framework called BayesRefine for program analyses expressed in Datalog, a popular declarative language for expressing program analyses [3, 7, 20, 29, 40, 42–44, 47]. BayesRefine automates the workflow of our approach from derivation extraction to the final constraint solving that removes incorrect generalization for Datalog-based analyses. We have evaluated its effectiveness with a thread-escape analysis and a datarace analysis [37] on a suite of Java programs, each comprising 95-369 KLOC. We apply BayesRefine to refine

[1]For conventional analyses, precision means true positive rate. For a Bayesian analyses, precision is not only determined by the alarms it derives, but also the ranking of these alarms.

```
1  readInputOdd()  {                          8
2      return input1(); //i1                  9  x1 = 1 / ( readInputOdd() + readInputEven() ); //i11,i21
3  }                                          10  y = readInputEven(); //i2y
4                                             11 x2 = 1 / y;
5  readInputEven()  {                         12 x3 = 1 / ( readInputOdd() - readInputEven() ); //i13,i23
6      return input2(); //i2                  13 x4 = -1 / y;
7  }                                          14 x5 = 1 / ( readInputEven() - readInputOdd() ); //i15,i25
```

Fig. 1. Example program.

abstractions produced by BinGraph, in the interactive alarm resolution use case of Bayesian analysis. BayesRefine improves BinGraph's performance on these two analyses by 18.86% and 33.01% respectively. The improvement on the datarace analysis can increase to 46.88% when the training set selection of BinGraph is not optimal.

In summary, the contributions of this paper are:

- We propose an online abstraction-search framework that is akin to CEGAR to solve the abstraction selection problem for Bayesian program analyses.
- We propose a technique that identifies the abstraction to refine by leveraging the theory of conditional independence.
- We evaluate our approach with two realistic analyses on large programs and demonstrate significant improvements to an existing offline learning-based abstraction selection technique.

2 Overview

This section introduces our approach informally with a graph reachability example, which reflects the core spirit of a "division-by-zero" analysis without exposing its complexity.

2.1 Example Program and the Dataflow Analysis Problem

Figure 1 shows an example program, which performs numerical computation using integers returned by invoking methods `readInputOdd` and `readInputEven`. These two methods are implemented by invoking library methods `input1` and `input2`. The method `input1` always generates odd numbers, while `input2` always generates even numbers. In line 9,12,14, both `readInputOdd` and `readInputEven` are called: At each line, after performing addition or subtraction using the two returned integers, the result is used as divisor in an integer division (in the assignment of variables `x1`, `x3` and `x5`). In line 10, only `readInputEven` is called and the returned integer is assigned to variable `y`, which is then used as a divisor in line 11 and line 14 when calculating `x2` and `x4` respectively. A dataflow analysis is applied to resolve potential "division-by-zero" alarms: For `i` $= 1, 2, 3, 4, 5$, there is a bug in the assignment of `xi` when the divisor can be zero, and we denote the corresponding potential alarm query as q_i. While alarms q_2 and q_4 are true, alarms q_1, q_3, and q_5 are false. This is because in the assignments to `x1`, `x3` and `x5`, the divisors are always odd numbers, while the value assigned to y by `readInputEven` in line 10 may be zero. Proving this fact requires correctly capturing specifications of the library methods `input1` and `input2`.

However, due to the source code being inaccessible (e.g., Windows APIs) or scalability concerns, it is very typical that program analyses cannot reason about all library methods. Here we assume the analysis does not know the specifications of `input1` and `input2`, and conservatively assumes they can return any number. As a result, the analysis works like a taint analysis and assumes that any number that is computed directed or indirectly using values returned (or "tainted") by either of the two library calls can be any number. Therefore it reports all five alarms.

To resolve the imprecision incurred by the unknown library specifications, we follow the previous approach [39] to convert the analysis into a Bayesian one by attaching probabilities, and boost

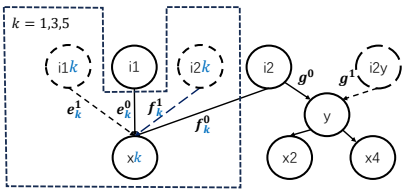
Input relations:

`edge`(u, v, k) (edge from node u to node v labeled k)
`abs`(k) (edge labeled k is allowed)
`taint`(u) (potential tainted dataflow from u)

Derived relations:

`path`(u, v) (node v is reachable from node u)
`alarm`(u) (potential division-by-0 alarm in node u)

Rules: (1): `path`(u, u).
(2): `path`$(u, v) :-$ `path`(u, w), `edge`(w, v, k), `abs`(k).
(3): `alarm`$(v) :-$ `path`(u, v), `taint`(u).



Input tuples: `edge(i1, x1,` e_1^0`), edge(i11, x1,` e_1^1`), ···, taint(i1), taint(i2), taint(i11), taint(i21)` ···
Abstraction tuples: `abs(`e_1^0`)` $\oplus$ `abs(`e_1^1`)`, ···, `abs(`f_1^0`)` $\oplus$ `abs(`f_1^1`)`, ···, `abs(`g^0`)` $\oplus$ `abs(`g^1`)`
Derived tuples: `path(i1, i1), path(i1, x1)`, ···, `path(i2, i2), path(i2, x1)`, ···, `alarm(x1)`, ···

Fig. 2. Datalog analysis for the graph reachability problem.

the analysis precision using user feedback. Briefly, the analysis produces a ranking of alarms by computing their confidence scores, and refines the ranking by asking the user to label the top-ranked alarm iteratively. Such interaction can continue as long as the user likes. In general, the earlier the true alarms are discovered, the better the user's experience is. For simplicity, we measure how many turns are needed to discover all true alarms.

Finally, the analysis is a selectively context-sensitive analysis which is parameterized by whether a `readInputOdd` or `readInputEven` invocation is inlined. Although no matter what abstraction produced by tweaking the context sensitivity cannot resolve the false alarms directly, we will show that they do affect how user feedback generalizes in the Bayesian analysis and demonstrate how our approach finds an appropriate abstraction on the fly.

2.2 The Graph Reachability Problem

For exposition purposes, we cast the dataflow analysis problem as a graph reachability problem, whose encoding is visualized in the graph shown in Figure 2. Node `i1` and `i2` represent invocations to `input1` and `input2` respectively, and node `y` represents variable `y`. Node `xk` represents variables `xk` ($k = 1, 2, 3, 4, 5$), where the subgraph in the dotted polygon repeats for `x1`, `x3`, and `x5`. The edges denote dataflows in the program and the analysis reports an alarm q_i if `i1` or `i2` can reach a variable node `xi`. Finally, the analysis is selectively context-sensitive in a way that it can choose to inline an invocation to `readInputOdd` or `readInputEven`, which effectively distinguishes different invocations to `input1` or `input2`. We represent an inlining as creating a copy of the corresponding node (e.g., `i11` for `i1` when the invocation to `readInputOdd` in the assignment to `x1` is inlined). We label the edges and use an array of labels in the form of $\{e_1^0$ or e_1^1, e_3^0 or e_3^1, e_5^0 or e_5^1, f_1^0 or f_1^1, f_3^0 or f_3^1, f_5^0 or f_5^1, g^0 or $g^1\}$ to denote the abstraction, which is a binary array of choices in inlining. Here the e and f edges denote flows from `input1` and `input2` to `x` variables respectively, the g edges denote the flow from `input2` to y, and the superscripts denote whether the corresponding method call is inlined. Figures 3 to 5 show the graphs under different abstractions where the edges are chosen based on the labels in the abstraction. Note that even though all alarms are derived under these abstractions, the connectivity between nodes is different.

Figure 2 shows the parametric analysis in Datalog, which takes tuples in input relations, and derives output tuples by computing the least fixed-point of the rules. In input relations, `taint`(u) represents that the dataflow from node u is tainted, which reflects the analysis' imprecision in library specifications. All the input relations are fixed except the `abs` relation, which specifies the analysis abstraction. Figure 2 lists the input tuples at the bottom and separately lists the possible choices of `abs` tuples where $\oplus$ denotes xor. The derived relation `path` contains pair (u, v), representing that node v is reachable from node u along a path with only edges whose labels appear in relation `abs`.

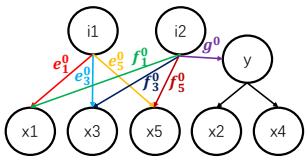


Fig. 3. Graph encoding under abs = $\{e_1^0, e_3^0, e_5^0, f_1^0, f_3^0, f_5^0, g^0\}$.

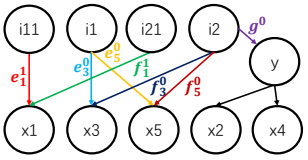


Fig. 4. Graph encoding under abs = $\{e_1^1, e_3^0, e_5^0, f_1^1, f_3^0, f_5^0, g^0\}$.

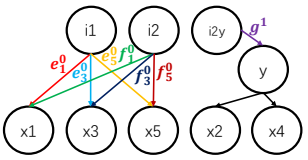


Fig. 5. Graph encoding under abs = $\{e_1^0, e_3^0, e_5^0, f_1^0, f_3^0, f_5^0, g^1\}$.

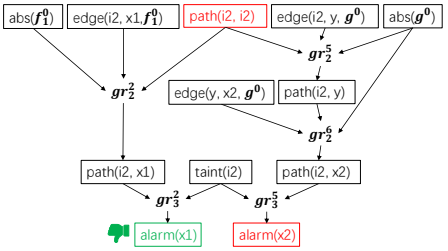


Fig. 6. A part of the derivation graph (abs = $\{e_1^0, e_3^0, e_5^0, f_1^0, f_3^0, f_5^0, g^0\}$).

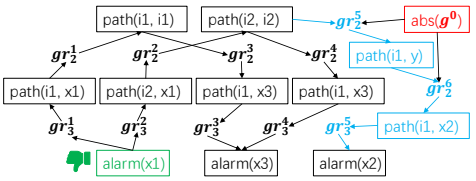


Fig. 7. The information-transmission graph starting from alarm(x1).

The ultimate goal of the analysis is to derive alarm tuples alarm(u) that represent that a potentially tainted dataflow reaches node u. The computation of alarm is expressed by rules (1),(2) and (3) all of which are Horn clauses with implicit universal quantification. Rule (1) states that each node is reachable from itself. Rule (2) states that if node w is reachable from node u and edge (w, v) is allowed, then node v is reachable from node u. Rule (3) states that if node v is reachable from node u and the dataflow from u is tainted, then there is a potential alarm in node v. The analysis will derive all five alarms no matter what abstraction is used as copies of i1 and i2 are also tainted.

2.3 The Bayesian Analysis and the Over-Generalization Problem

We next try to address the imprecision by converting the analysis into a Bayesian one so that the user can prioritize inspecting true alarms, and demonstrate how it is failed by an inappropriate abstraction. The Bayesian analysis is implemented as a Bayesian network, which is constructed from the derivation of the above Datalog analysis.

To start, we apply the cheapest abstraction abs = $\{e_1^0, e_3^0, e_5^0, f_1^0, f_3^0, f_5^0, g^0\}$ where no method is inlined, which is a default choice in many cases. Next we construct a derivation graph that shows reasoning steps to derive all alarms under abs. For simplicity, we denote alarm(xi) as q_i ($i = 1, 2, \cdots, 5$). Figure 6 shows the key part of the graph to derive q_1 and q_2. Each instantiation of a rule is encoded as a ground rule node. For instance, ground rule node gr_2^2 represents path(i2,x1) :- path(i2,i2), edge(i2,x1,f_1^0), abs(f_1^0), and gr_3^2 represents alarm(x1) :- path(i2,x1), taint(i2). Then, we convert the derivation graph into a Bayesian network following the previous work [39]: We do it by quantifying the incompleteness of each rule as a probability. In practice, one can learn these probabilities, but for illustration, we assign 0.99 to all the rules here. We use universally quantified variables gr_2 and gr_3 to denote ground rule nodes of rule (2) and (3) respectively, and the conditional probabilities are (Note all variables below are universally quantified):

$$\Pr(gr_2 \mid \texttt{path}(u,w) \wedge \texttt{edge}(w,v,k) \wedge \texttt{abs}(k)) = 0.99 \qquad \Pr(\texttt{path}(u,u)) = 0.99$$

$$\Pr(gr_2 \mid \neg\,\texttt{path}(u,w) \vee \neg\,\texttt{edge}(w,v,k) \vee \neg\,\texttt{abs}(k)) = 0$$

$$\Pr(gr_3 \mid \texttt{path}(u,v) \wedge \texttt{taint}(u)) = 0.99 \qquad \Pr(gr_3 \mid \neg\,\texttt{path}(u,v) \vee \neg\,\texttt{taint}(u)) = 0$$

Rank	Alarm	Prob.
1	q_1	0.999
2	q_3	0.999
3	q_5	0.999
4	q_2	0.961
5	q_4	0.961

Fig. 8. Original ranking.

Rank	Alarm	Prob.
1	q_3	0.878
2	q_5	0.878
3	q_2	0.644
4	q_4	0.644

Fig. 9. Ranking after inspecting q_1 with abs = $\{e_1^0, e_3^0, e_5^0, f_1^0, f_3^0, f_5^0, g^0\}$.

Rank	Alarm	Prob.
1	q_2	0.961
2	q_4	0.961
3	q_3	0.878
4	q_5	0.878

Fig. 10. Ranking after inspecting q_1 with abs = $\{e_1^0, e_3^0, e_5^0, f_1^0, f_3^0, f_5^0, g^1\}$.

And for tuples $\texttt{path}(u, v)$ and $\texttt{alarm}(u^*)$, we suppose that all the ground rule nodes which have an directed edge pointing to it are $gr_2^1, gr_2^2, \ldots, gr_2^t$ and $gr_3^1, gr_3^2, \ldots, gr_3^s$ respectively, then we have:

$$\Pr(\texttt{path}(u,v) \mid gr_2^1 \vee gr_2^2 \vee \cdots \vee gr_2^t) = 1 \qquad \Pr(\texttt{path}(u,v) \mid \neg gr_2^1 \wedge \neg gr_2^2 \wedge \ldots \wedge \neg gr_2^t) = 0$$

$$\Pr(\texttt{alarm}(u^*) \mid gr_3^1 \vee gr_3^2 \vee \cdots \vee gr_3^s) = 1 \qquad \Pr(\texttt{alarm}(u^*) \mid \neg gr_3^1 \wedge \neg gr_3^2 \wedge \ldots \wedge \neg gr_3^s) = 0$$

At first, the Bayesian program analysis calculates the confidence value $\Pr(q_i)$ for each alarm q_i, and the ranking of alarms ordered by it is shown in Figure 8. Next the user inspects the top alarm q_1 and labels it to be false. Then Bayesian inference is performed to compute the probabilities of other alarms $\Pr(q_i \mid \neg q_1)$ conditioned on $\neg q_1$. Figure 9 shows the new ranking according to the posterior probabilities. Notice under the cheapest abstraction the five alarms are closely related in the Bayesian inference network (see Figure 3), and hence their confidence values all drop. After that, the user needs to inspect false alarms q_3 and q_5 before the last two true alarms are observed. As a result, in order to discover all true alarms, the user needs to inspect all the three false alarms.

This result is due to the over-generalization: Under abstraction $\{e_1^0, e_3^0, e_5^0, f_1^0, f_3^0, f_5^0, g^0\}$, the observation on false alarm q_1 not only degrades the probabilities of other correlated false alarms (e.g. q_3 and q_5), but also has an undesired generalization to true alarms (e.g. q_2 and q_4). In Figure 6, we can see that the negative feedback excessively blames the common ancestor of alarms `path(i2,i2)`, as Bayesian analysis decreases its posterior probability which in turn leads to the drop of confidence values of its descendants including those true alarms. On the other hand, under abstraction $\{e_1^0, e_3^0, e_5^0, f_1^0, f_3^0, f_5^0, g^1\}$, the negative feedback on q_1 only affects the false alarms (see Figure 5), which enables the user to inspect the true alarms q_2 and q_4 first.

2.4 Our Approach

Our solution is to perform abstraction refinement to alleviate undesired connections between true and false alarms in the Bayesian networks. Notice that there are 128 abstractions in total, each involving a different choice of the zero/one versions of edge labels. Our goal is to choose proper labels that removes undesired generalizations. For instance, changing e_1^0 to e_1^1 will not break our aimed connection as shown in Figures 3 and 4. Besides, we also wish to minimize the cost of the refined abstraction for efficiency. Abstractions with more zero versions of edge labels are cheaper.

Our approach is inspired by iterative counterexample-guided abstraction refinement (CEGAR), but the technical details are drastically different. It first samples alarms for the user to inspect to determine if the current generalization is correct, then constructs a graph to reflect information transmission on the Bayesian network, and finally removes incorrect generalizations by solving a *maximum satisfiability* (Max-SAT) problem. We next explain each step in detail.

After receiving the negative feedback on q_1 and updating the ranking, to see whether its generalization is desired, we ask the user to inspect k unobserved alarms whose ranks drop the most. Here we focus on rank drops because after getting negative feedback, the increase in an alarm's rank is typically caused by drops in other alarms' probabilities. The chosen alarms also exclude ones that are not connected to q_1 as their probabilities are not affected by the feedback. In this example, we set $k = 2$. We suppose the program in Figure 1 is excerpted from a larger program,

and there are other "division-by-zero" alarms but they are not related to methods `readInputOdd` and `readInputEven`. Therefore, their confidence values are not affected by the user feedback, and ranks of q_i ($i = 2, 3, 4, 5$) drop compared with them. Then we ask the user to inspect q_2 and q_3 since the order between $q_2 - q_5$ remains unchanged as shown in Figures 8 and 9. Next we divide those checked alarms into two sets depending on whether their confidence values change correctly based on their ground truth. Here q_2 is true, but its confidence value dropped, so it falsely changes. Such false generalizations are similar to counterexamples in conventional CEGAR. On the other hand, the confidence value of false alarm q_3 drops, so we can say that it correctly changes. We hope to preserve such correct generalizations, which has no counterparts in conventional CEGAR.

Our key idea is to remove the undesired generalizations through refining the abstraction in the hope that other unchecked false generalizations are also removed while keeping correct generalizations. Concretely, we construct a graph $\mathcal{G}$ atop the original Bayesian network, which shows the transmission of posterior information from the label on the recently observed alarm. The graph is constructed based on the theory of *conditional independence*. For details the reader can refer to Section 5, while in the rest of this section we will describe final steps in our example. Figure 7 shows a simplified version of our constructed $\mathcal{G}$ for the example program, where all the input tuples except $\texttt{abs}(g^0)$ are removed. All we need to do is to separate the source alarm from those whose ranks falsely change while preserving connections to those whose ranks correctly change. In this example, it is to separate q_1 from q_2 and keep connections between q_1 and q_3. It can be realized by removing any node on the path from $\texttt{path(i2,i2)}$ to q_2 (**the blue path** in Figure 7), and the only related abstraction tuple is g^0. Besides, we also want to minimize the set of edge labels that need to refine for efficiency. This problem is similar to a minimum cut problem on a graph with connectivity constraints, a NP-hard problem [15]. We solve it by casting it as a Max-SAT problem.

By solving the Max-SAT problem, we change g_0 to g_1. Using this refined abstraction $\texttt{abs} = \{e_1^0, e_3^0, e_5^0, f_1^0, f_3^0, f_5^0, g^1\}$, we re-calculate the posterior probability of each alarm and get a new ranking as in Figure 10. Under this new abstraction, because the connections between true alarms $\{q_2, q_4\}$ and false alarms $\{q_3, q_5\}$ are cut off, the observation on $\{q_2, q_4\}$ will not affect $\{q_3, q_5\}$. Therefore, both true alarms are ranked higher than the false alarms. Although the user needs to inspect q_3 during sampling, the number of observed false alarms reduces from 3 to 2.

In this way, given a single unexpected generalization from the user's observation on a single alarm, our approach refines abstraction to reduce the user's alarm inspection burden of false alarms compared to original Bayesian program analysis.

3 Preliminaries

This section introduces necessary notations for describing our approach. We implement Bayesian program analysis by converting a Datalog-based analysis into a Bayesian network. For simplicity, we have limited our discussion to the original Datalog, but there should be no difficulty in expanding to its variants, such as Flix[29] and Formulog[3], as their proof structures are still graphs. Next, we will introduce the syntax and semantics of a Datalog program, a Datalog analysis with parametric abstractions, and how to convert a Datalog analysis into a Bayesian network.

3.1 Syntax and Semantics of Datalog

Figure 11(a) shows the syntax of Datalog. A set of input tuples $T_I \subseteq \mathbb{T}$ and a set of derivation rules $C \subseteq \mathbb{C}$ form a Datalog program $\mathcal{D} = (T_I, C)$. Each derivation rule has a head consisting of one literal (the result), and a body consisting of a list of literals (the condition). A literal is a relation name r followed by a list of arguments. We refer to a literal with only constants as a *tuple*. We define $\sigma \in \Sigma = \mathbb{L} \mapsto \mathbb{T}$ to be a *substitution function* mapping a literal $l = r(a_1, a_2, \ldots, a_n)$ to a tuple $\sigma(l) = r(d_1, d_2, \ldots, d_n)$ where there exists $\sigma' \in \mathbb{V} \mapsto \mathbb{D}$ such that $d_i = a_i$ if $a_i \in \mathbb{D}$ else $d_i = \sigma'(a_i)$.

$$(\textit{Datalog Program})\quad \mathcal{D} ::= (T_I, C) \qquad (\textit{Rule})\quad c ::= l :\text{-}\ \bar{l} \in \mathbb{C} \qquad (\textit{Literal})\quad l ::= r(\bar{a}) \in \mathbb{L}$$
$$(\textit{Argument})\quad a ::= d \mid v \qquad (\textit{Tuple})\quad t ::= r(\bar{d}) \in \mathbb{T} \qquad (\textit{Ground Rule})\quad gc ::= t :\text{-}\ \bar{t}$$

(a) Syntax of Datalog

$$(\textit{Variable})\quad v \in \mathbb{V} \qquad (\textit{Constant})\quad d \in \mathbb{D} \qquad (\textit{Relation Name})\quad r \in \mathbb{R}$$
$$(\textit{Alarm})\quad q \in \mathbb{Q} \subseteq \mathbb{T} \qquad (\textit{Abstraction})\quad A \in \mathbb{A} \subseteq \mathcal{P}(\mathbb{T}) \qquad (\textit{Substitution})\quad \sigma \in \mathbb{L} \mapsto \mathbb{T}$$
$$(\textit{Input tuples})\quad T_I \in \mathcal{P}(\mathbb{T}) \qquad (\textit{Derivation rules})\quad C \in \mathcal{P}(\mathbb{C})$$

(b) Auxiliary definitions

$$F_C, f_c \in \mathcal{P}(\mathbb{T}) \to \mathcal{P}(\mathbb{T}) \qquad F_C(T) = T \cup \{f_c(T) \mid c \in C\}$$
$$f_{l_0 :\text{-} l_1,\ldots,l_n}(T) = \{\sigma(l_0) \mid \sigma(l_i) \in T \text{ for } 1 \le i \le n, \sigma \in \Sigma\}$$

(c) Semantics of Datalog

Fig. 11. Syntax and semantics of Datalog.

Figure 11(c) describes the semantics of Datalog. Specifically, function $f_{l_0 :\text{-} l_1,\ldots,l_n}$ derives a tuple from known tuples - if there exists a substitution σ such that $\sigma(l_1), \sigma(l_2), \cdots, \sigma(l_n)$ are all known, $\sigma(l_0)$ is derived. Function F_C computes output tuples by applying f_c for each rule c in C to a given set of tuples for one round. Then we define the output tuples of a Datalog program $\mathcal{D}$ as $[\![\mathcal{D}]\!] \subseteq \mathbb{T}$, that is obtained by repeatedly applying F_C. Concretely, the initial T is set to T_I and we iteratively change T to $F_C(T)$ until reaching the least fixpoint $T_f = F_C(T_f)$, at which point T_f is $[\![\mathcal{D}]\!]$.

The above derivation process can be modeled as a *derivation graph* $G(\mathcal{D}) = [V, E]$. First by replacing all literals with all possible output tuples in $[\![\mathcal{D}]\!]$, each rule is transformed to a set of ground rules. The transformation from rule $c = l_0 :\text{-}\ l_1, \ldots, l_n$ to ground rule $gc = t_0 :\text{-}\ t_1, \ldots, t_n$ is possible when there exists a *substitution function* $\sigma \in \Sigma$ and $\sigma(l_i) = t_i \in [\![\mathcal{D}]\!]$ holds for $i = 0, 1, \ldots, n$. Then we denote the set of all ground rules as $GC(\mathcal{D})$, and define the node set V of *derivation graph* G as $GC(\mathcal{D}) \cup [\![\mathcal{D}]\!]$. At last, for each $gc = t_0 :\text{-}\ t_1, \ldots, t_n \in GC(\mathcal{D})$, we construct $n + 1$ directed edges $(t_1, gc), \ldots, (t_n, gc), (gc, t_0)$ in E, and hence complete the *derivation graph* $G(\mathcal{D})$.

Example. Take the incomplete derivation graph in Figure 6 as instance: A ground rule is `path(i2,x1) :- edge(i2, x1,` f_1^0`), path(i2,i2), abs(`f_1^0`)`, and gr_2^2 is its corresponding node in $G(\mathcal{D})$.

3.2 Datalog Program Analysis with Parametric Abstractions

For a Datalog program implementing an analysis with parametric abstractions, its input consists of two parts: (1) tuples that encode the program being analyzed, (2) tuples that determine the degree of program abstraction. We refer to the set containing all input tuples in (2) as an **abstraction**. Only **abstractions** will be changed by our approach, while the remaining input tuples will not.

Example. In Figure 2, for the Datalog encoding of our graph reachability problem, the input tuples of `taint` and `edge` belong to (1), while the input tuples of the `abs` relation belong to (2).

An alarm $q \in \mathbb{Q}$ is a special kind of tuple that represents a report for a bug or an undesirable program property. A set of alarms Q can be divided into two sets $(\mathcal{R}(Q), \mathcal{I}(Q))$, representing false alarms and true alarms respectively. In other words, the bugs or properties described by $\mathcal{R}(Q)$ actually do not arise during the execution of the given program. The output tuples of Datalog program $\mathcal{D}$ based on an abstraction $A \in \mathbb{A}$ is defined as $[\![\mathcal{D}, A]\!]$. Suppose $\boldsymbol{Q}$ contains all possible alarms, then the output alarms of $\mathcal{D}$ using A are $\boldsymbol{Q} \cap [\![\mathcal{D}, A]\!]$. Suppose the analysis is always sound, then when changing the abstraction A, the true alarm part $\mathcal{I}(\boldsymbol{Q} \cap [\![\mathcal{D}, A]\!])$ will remain the same.

Example. The set of alarms in our graph reachability example is $\boldsymbol{Q}$ where $\mathcal{R}(\boldsymbol{Q}) = \{$`alarm(x1)`, `alarm(x3)`, `alarm(x5)`$\}$, $\mathcal{I}(\boldsymbol{Q}) = \{$`alarm(x2)`, `alarm(x4)`$\}$.

To efficiently search for a good abstraction, abstraction selection approaches for conventional analyses typically assume there exists a precision preorder $\sqsubseteq$ on the family of abstractions. How

to define this preorder depends on the analysis but it should satisfy that a more precise analysis would not produce more false alarms. In our case, we encode this precision preorder as the size order of parameters in abstractions. Take our graph reachability example for instance, in the family $\mathbb{A}$ of valid abstractions, $A_1 = \{\mathsf{abs}(e_1^{i_1}), \mathsf{abs}(e_3^{i_2}), \mathsf{abs}(e_5^{i_3}), \mathsf{abs}(f_1^{i_4}), \mathsf{abs}(f_3^{i_5}), \mathsf{abs}(f_5^{i_6}), \mathsf{abs}(g^{i_7})\}$ and $A_2 = \{\mathsf{abs}(e_1^{j_1}), \mathsf{abs}(e_3^{j_2}), \mathsf{abs}(e_5^{j_3}), \mathsf{abs}(f_1^{j_4}), \mathsf{abs}(f_3^{j_5}), \mathsf{abs}(f_5^{j_6}), \mathsf{abs}(g^{j_7})\}$. They are ordered as follows: $A_1 \sqsubseteq A_2$ if and only if $i_k \leq j_k$ $(\forall k \in \{1, 2, \cdots, 7\})$. With this encoding, for an abstraction A and a subset of it $A_{select} \subseteq A$, we can find a more precise abstraction A' by only changing parameters in tuples of A_{select}. As for the main focus of this work, learnability and generalization, we will give a vital assumption about them on $\sqsubseteq$ in Section 5.2.

3.3 Bayesian Program Analysis with Parametric Abstractions

We make a conventional analysis Bayesian by converting its derivation graph into a Bayesian network. For Datalog program $\mathcal{D}$ equipped with abstraction A, its ground rules and *derivation graph* are defined as $GC(\mathcal{D}, A)$ and $G(\mathcal{D}, A)$ respectively. To convert $G(\mathcal{D}, A)$ into a Bayesian network, we follow previous works [11, 19, 21, 39] and apply an approximation which removes ground rules that can form cycles. In the rest of the paper, we assume $G(\mathcal{D}, A) = [V, E]$ is acyclic.

When converting a *derivation graph* to a Bayesian network, a Bernoulli random variable $x(v)$ is created for each v in V. Then we attach a probability to each rule in the original Datalog program $\mathcal{D}$, indicating how likely the rule holds. We obtain such probabilities through a training process using the expectation maximization algorithm [24]. For a ground rule $gc = t_0 \text{ :- } t_1, \ldots, t_n$, we assume its corresponding rule is assigned with probability p and create edges with conditional probabilities $\Pr(x(gc) \mid x(t_1) \wedge x(t_2) \wedge \cdots \wedge x(t_n)) = p$ and $\Pr(x(gc) \mid \neg x(t_1) \vee \neg x(t_2) \vee \cdots \vee \neg x(t_n)) = 0$. For each tuple $t \in [\![\mathcal{D}, A]\!]$, let all the ground rules that can derive it be $gc_1, gc_2, \ldots, gc_n$, we create edges between their corresponding random variables with conditional probabilities $\Pr(x(t) \mid x(gc_1) \vee x(gc_2) \vee \cdots \vee x(gc_n)) = 1$ and $\Pr(x(t) \mid \neg x(gc_1) \wedge \neg x(gc_2) \wedge \cdots \wedge \neg x(gc_n)) = 0$.

Finally, we introduce interactive alarm resolution, the specific use case of Bayesian program analysis considered in this paper. In this application, the analysis system initially uses a Datalog program $\mathcal{D}$ with an abstraction A_0 to derive a set Q as possible alarms. Then after converting the *derivation graph* $G(\mathcal{D}, A_0)$ to a Bayesian network, the system will interact with the user for multiple rounds. In each round, the analysis presents an alarm to the user and receives a binary feedback, indicating whether the alarm is a true positive. In the ith-round, the Bayesian analysis system employs a probability inference algorithm [36] on the Bayesian network to calculate the marginal probability of each alarm, and presents the alarm with the highest probability, $a = \arg\max_{t \in Q} \Pr\left(x(t) \mid \bigwedge_{j=1,2,\ldots,i-1} e_j\right)$ given existing user feedback. Here e_i represents the received user feedback at the ith-round: If a is true then $e_i = x(a)$ else $e_i = \neg x(a)$. The user feedback is added as an evidence to the Bayesian network. The interaction can continue as long as the user wants. In general, the earlier the true positives are presented, the better the user's experience is. The user can pick different termination conditions in practice (e.g., when a given number of consecutive false positives are reported). But in an experimental setting, one way to measure the performance of such an interactive analysis is to count the number of interactions that is required to discover all the true positives (denoted by $\mathcal{I}(Q)$). This is the main metric we will use in the paper.

4 Problem Definition

With notations in Section 3, we define our problem formally as below:

Definition 4.1 (Optimum Abstraction Problem of Bayesian Program Analysis). Given a Datalog program $\mathcal{D}$ parameterized with an abstraction family $(\mathbb{A}, \sqsubseteq)$, a set Q which consists of all

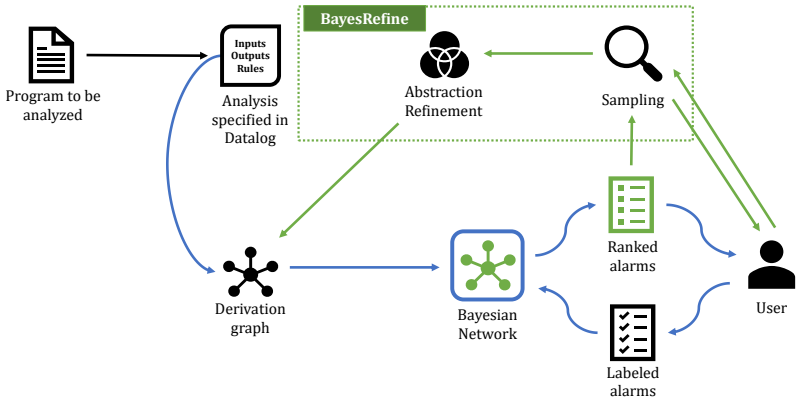


Fig. 12. Overall workflow of BAYESREFINE

its possible alarms, the optimum abstraction problem in interactive alarm resolution is to find an $A \in \mathbb{A}$ under which the number of interactions to identify all true alarms ($\mathcal{I}(Q)$) is the smallest.

A natural idea is to use traditional abstraction selection approaches, which try to search for precise yet efficient abstractions. However, the premise of it is that the more precise the abstraction is, the better the generalization in Bayesian analysis is. This assumption has been invalidated in the BinGraph paper [53] as a too fine-grained abstraction can unnecessarily break the correlations between program facts, and lead to overfitting. For example, inlining every method invocation in the example in Section 2 would render user feedback useless. BinGraph addresses the problem using a learning-based approach but is heavily affected by the training set selection and can yield sub-optimal performance on new programs. We next present our approach, which tunes the abstraction on the fly by analyzing why the current abstraction causes false generalization.

5 Our Framework

Our framework, BayesRefine, refines abstractions based on the effect of generalization during interactions, whose high-level workflow and overall algorithm are presented in Section 5.1. It achieves high automation while remaining efficient for two main ideas. The first is to model the problem of finding a refined abstraction as cutting and preserving information transmission in a Bayesian network (described in Section 5.2). The second is to reduce that problem further into Max-SAT (described in Section 5.3). We have implemented BayesRefine atop BINGO, which performs Bayesian network inference by using loopy belief propagation implemented in LibDAI [34].

5.1 Workflow

Figure 12 shows the workflow of our framework, BayesRefine. Our framework takes as input a program and a conventional analysis specified in Datalog that is parameterized by an abstraction family $(\mathbb{A}, \sqsubseteq)$. It first runs the analysis with an initial abstraction $A_0 \in \mathbb{A}$ and converts the analysis derivation on the graph into a Bayesian network following the procedure described in Section 3.3. The initial abstraction A_0 can be the coarsest abstraction in the abstraction family or an abstraction produced by an offline selection method such as BinGraph [53].

Next, our framework performs interactive alarm resolution as described in Section 3.3. In each interaction, the alarm with the highest probability is presented to the user for inspection. The user then indicates whether it is true, and this posterior information is incorporated as evidence in the following iterations. This interaction process is illustrated as **the blue arrows** in the bottom-right corner of Figure 12. In addition, we introduce a module for abstraction refinement in our framework

Algorithm 1 Counterexample-based Abstraction Refinement Algorithm.

Require: $(\mathcal{D}, A_0, \boldsymbol{Q})$, where $\mathcal{D}$ is the Datalog-expressed program analysis, A_0 is the initial abstraction, and $\boldsymbol{Q}$ is the set of unsolved alarms.

1: **procedure** BayesRefine($\mathcal{D}$, A_0, $\boldsymbol{Q}$)
2: $A \leftarrow A_0, Q \leftarrow \boldsymbol{Q}, Q_{observed} \leftarrow \emptyset, \bar{e} \leftarrow \emptyset$, *count* $\leftarrow 0$
3: **while** the termination condition is not satisfied **do**
4: $L \leftarrow [q_1, q_2, \ldots, q_n]$, where $Q = \{q_1, q_2, \ldots, q_n\}$ and $\forall 1 \le i < j \le n$, $\Pr\left(x(q_i) \mid \bar{e}\right) \ge \Pr\left(x(q_j) \mid \bar{e}\right)$ (L is a list of Q ranked by alarms' probabilities before user's feedback)
5: $e_{temp} \leftarrow$ user's inspection of q_1, $\bar{e} \leftarrow \bar{e} \wedge e_{temp}$, $Q \leftarrow Q \backslash \{q_1\}$, $Q_{observed} \leftarrow Q_{observed} \cup \{q_1\}$
6: *count* $\leftarrow 0$ if q_1 is a true alarm, *count* $\leftarrow$ *count* $+ 1$ if q_1 is a false alarm
7: $L^* \leftarrow [q_1^*, q_2^*, \ldots, q_{n-1}^*]$, where $Q = \{q_1^*, q_2^*, \ldots, q_{n-1}^*\}$ (which is equal to the set $\{q_2, q_3, \ldots, q_n\}$) and $\forall 1 \le i < j \le n-1$, $\Pr\left(x(q_i^*) \mid \bar{e}\right) \ge \Pr\left(x(q_j^*) \mid \bar{e}\right)$ (L^* is a list of Q ranked by alarms' probabilities after user's feedback)
8: **if** *count* $> k_1$ **then**
9: $\bar{q} = \{q'_1, q'_2, \ldots, q'_{k_2}\}$, where $\bar{q}$ is the set of k_2 alarms of Q which have largest rank drops from L to L^*.
10: $e_{sample} \leftarrow$ user's inspection of $\bar{q}$, $\bar{e} \leftarrow \bar{e} \wedge e_{sample}$, $Q \leftarrow Q \backslash \bar{q}$
11: $(A, \mathit{ifRefined}) \leftarrow$ Refine($q_1, \bar{q}, Q_{observed}, \mathcal{D}, A, \bar{e}$)
12: $Q_{observed} \leftarrow Q_{observed} \cup \bar{q}$, $Q \leftarrow Q \cap [[\mathcal{D}, A]]$
13: **if** *ifRefined* **then** *count* $\leftarrow 0$

(**the green arrows** in Figure 12). It samples several alarms for the user to inspect, and then checks if changes of their probabilities incurred by the newly added evidence are in the correct direction according to their ground truth. Instructed by this information, the module performs abstraction refinement. Then with the newly produced abstraction, the interaction goes to the next iteration by rerunning the analysis to update the Bayesian network.

Our main algorithm (Algorithm 1) formally describes the iterative abstraction refinement part. The set of unsolved alarms $\boldsymbol{Q}$ contains all possible alarms derived using the initial abstraction A_0. We denote the set of alarms that have been inspected by the user as $Q_{observed}$ and the corresponding evidence as $\bar{e}$. We use a variable *count* to record the number of **false alarms that have been continuously observed**, and it is initialized as 0. Later, the abstraction refinement will be only triggered when *count* exceeds a threshold k_1. Intuitively, if the user keeps seeing true alarms, it means the Bayesian analysis is predicting correctly and there needs no adjustment. On the other hand, a large *count* indicates the user feedback is generalizing badly and the abstraction needs to be adjusted. Moreover, the threshold controls the frequency of abstraction refinement as applying it too frequently may affect the overall efficiency.

In Line 4, for each q in Q, $x(q)$ denotes the Bernoulli random variable representing whether q is a true alarm. Then in Lines 4 to 7, the user inspects the alarm with the highest probability and the algorithm generates two lists L and L^* ranked by the alarms' probabilities before and after the user's feedback. If *count* exceeds the threshold k_1, we will sample k_2 alarms which have the largest rank drops conditioned on user's inspection in Line 9, since they are affected most by the negative feedback provided by the user. Next these sampled alarms are inspected by the user and removed from set Q, and the evidence $\bar{e}$ will be modified according to user's feedback in Line 10. After that, in Line 11 the algorithm uses the sampled alarms $\bar{q}$, observed alarms $Q_{observed}$ and the recently inspected alarm q_1 to perform abstraction refinement in function Refine. Finally, in Line 12 the algorithm adds alarms observed in this iteration to $Q_{observed}$, and alarms that cannot be derived

Algorithm 2 Function Refine

Require: $q_1, \bar{q}, Q_{observed}, \mathcal{D}, A, \bar{e}$
1: $\mathcal{G}(G(\mathcal{D}, A), Q_{observed}, q_1) \leftarrow$ GenInfoTransGraph$(\mathcal{D}, A, Q_{observed}, q_1, \bar{e})$
2: $p \leftarrow$ GenMaxSAT$(\mathcal{G}(G(\mathcal{D}, A), Q_{observed}, q_1), \bar{q}, q_1, \bar{e})$
3: $(r, A_{refined}) \leftarrow$ solve(p)
4: **if** r == Satisfiable **then return** $(A_{refined}, true)$
5: **else return** $(A, false)$

through Datalog inference under the new abstraction are also removed. After that, our algorithm starts the next iteration. The user can pick various termination conditions in practice, and in our experimental settings, we choose when all true alarms($\mathcal{I}(Q)$) have been observed for evaluation purpose.

The core of our approach is function Refine, which is outlined in Algorithm 2. It can be divided into two stages: constructing a information-transmission graph that reflects generalization of posterior information among facts in Line 1, and refining the abstraction by solving a Max-SAT problem that aims at removing undesired generalization and preserving desired generalization on the graph in Line 2 - Line 5. We describe the two stages in detail in the following subsections.

5.2 Information-Transmission Graphs

In order to model the generalization effect in a Bayesian inference network, we refer to the theory of conditional independence, which can be illustrated using three nodes forming a line. Suppose node Z is already observed, will observing node X change the marginal probability of node Y?

The theory of **D-separation** answers this question [4]. Suppose a Bayesian network is represented as a directed graph G, we define a **generalized path** on G as a sequence of edges forming a path on its undirected counterpart. On a generalized path, the arrows meet either head-to-tail, tail-to-head, head-to-head or tail-to-tail at nodes. When the posterior information on a node u can reach another node v through a **generalized path**, the path is called **unblocked** for (u, v). Consider such path through nodes x, z, y in order, whether it is unblocked is summarized below:

- For $x \rightarrow z \rightarrow y$, it is unblocked if and only if z is not observed.
- For $x \leftarrow z \leftarrow y$, it is unblocked if and only if z is not observed.
- For $x \leftarrow z \rightarrow y$, it is unblocked if and only if z is not observed.
- For $x \rightarrow z \leftarrow y$, it is unblocked if and only if z or a z's descendant **is observed**, where a node v is z's descendant if and only if there exists a directed path from z to v.

By extending the situation beyond three nodes, we have the following formal definition [24]:

Definition 5.1 (D-Separation). **A**, **B**, and **C** are non-intersecting subsets of nodes in a directed graph. A generalized path from **A** to **B** is blocked by **C** if it contains a node such that (1) the arrows on the path meet either head-to-tail, tail-to-head or tail-to-tail at the node, and the node is in the set **C**, or (2) the arrows meet head-to-head at the node, and neither the node, nor any of its descendants, are in the set **C**. If all paths from **A** to **B** are blocked by **C**, **A** is said to be D-separated from **B** by **C**.

Example. In Figure 6 let **A** = {`alarm(x1)`}, **B** = {`path(i2, x2)`} and **C** = $\emptyset$, the generalized path `alarm(x1)` $\leftarrow gr_3^2 \leftarrow$ `taint(i2)` $\rightarrow gr_3^5 \leftarrow$ `path(i2, x2)` is blocked, since gr_3^5 and its descendant `alarm(x2)` are both not in **C**. But if we add `alarm(x2)` to **C**, this path becomes unblocked, which means by observing `alarm(x2)` the posterior information from **A** can reach **B** through this path.

Based on this definition, we let **C** be the set of observed alarms $Q_{observed}$, and propose a vital assumption about the precision preorder of abstractions $\sqsubseteq$:

Assumption 1. For a pair of abstractions (A_1, A_2) which satisfies $A_1 \sqsubseteq A_2$ (it means that A_2 is more precise than A_1), and for any pair of nodes (u, v) representing alarms, if $\{u\}$ is D-separated from $\{v\}$ by any node set $\mathbf{C}$ in $G(\mathcal{D}, A_1)$, $\{u\}$ will be D-separated from $\{v\}$ by $\mathbf{C}$ in $G(\mathcal{D}, A_2)$.

In other words, if two alarms are D-separated by previously observed alarms, they will remain D-separated using a more precise abstraction. Such an assumption guarantees that when we increase the precision of the analysis, it will not increase the degree of undesired generalization, which is vital to guarantee the progression of our approach. In practice, the conventional preorder defined in Section 3.2 usually satisfies the requirement, so there is no additional effort to change it.

If a generalized path from the recently inspected alarm $\{q_1\}$ to a node set $\mathbf{B}$ is blocked by $\mathbf{C}$, then the posterior information cannot reach nodes in $\mathbf{B}$ through that path. Inspired by it, we model the generalization on a derivation graph $G(\mathcal{D}, A) = [V, E]$ as reachability on a directed information-transmission graph $\mathcal{G}(G, \mathbf{C}, q_1) = [V, \mathscr{E}]$ (Note that the two graphs have the same set of nodes V). We write $\mathcal{G}(G, \mathbf{C}, q_1)$ as $\mathcal{G}$ for brevity.

We define an ordered pair $\langle u, v\rangle$ as an edge $u \longrightarrow v$ in E or $\mathscr{E}$, and we mark all nodes in V with 3 numbers: $\mathbf{mark}(v) = 0$ if and only if v is not observed and does not have an observed descendant; $\mathbf{mark}(v) = 1$ if and only if v is observed; $\mathbf{mark}(v) = 2$ if and only if v is not observed but has an observed descendant. The edge set $\mathscr{E}$ could be constructed from E by calculating the least fixpoint of following translation rules: ($\mathscr{E}$ is firstly set to be $\emptyset$ and suppose $V = \{v_0, v_1, \cdots, v_n\}$, $v_0 = q_1$)

$$\{\langle v_0, v_i\rangle\} \subseteq E \vee \{\langle v_i, v_0\rangle\} \subseteq E \Rightarrow \mathscr{E} \coloneqq \mathscr{E} \cup \{\langle v_0, v_i\rangle\} \quad (\forall\, i \in \mathbb{N}^*) \tag{1}$$

$$\left\{\left\langle v_i, v_j\right\rangle, \left\langle v_j, v_k\right\rangle\right\} \subseteq E \wedge \left\{\left\langle v_i, v_j\right\rangle\right\} \subseteq \mathscr{E} \wedge \mathbf{mark}(v_j) \neq 1 \Rightarrow \mathscr{E} \coloneqq \mathscr{E} \cup \left\{\left\langle v_j, v_k\right\rangle\right\} \quad (\forall\, i, j, k \in \mathbb{N}^*) \tag{2}$$

$$\left\{\left\langle v_j, v_i\right\rangle, \left\langle v_k, v_j\right\rangle\right\} \subseteq E \wedge \left\{\left\langle v_i, v_j\right\rangle\right\} \subseteq \mathscr{E} \wedge \mathbf{mark}(v_j) \neq 1 \Rightarrow \mathscr{E} \coloneqq \mathscr{E} \cup \left\{\left\langle v_j, v_k\right\rangle\right\} \quad (\forall\, i, j, k \in \mathbb{N}^*) \tag{3}$$

$$\left\{\left\langle v_j, v_i\right\rangle, \left\langle v_j, v_k\right\rangle\right\} \subseteq E \wedge \left\{\left\langle v_i, v_j\right\rangle\right\} \subseteq \mathscr{E} \wedge \mathbf{mark}(v_j) \neq 1 \Rightarrow \mathscr{E} \coloneqq \mathscr{E} \cup \left\{\left\langle v_j, v_k\right\rangle\right\} \quad (\forall\, i, j, k \in \mathbb{N}^*) \tag{4}$$

$$\left\{\left\langle v_i, v_j\right\rangle, \left\langle v_k, v_j\right\rangle\right\} \subseteq E \wedge \left\{\left\langle v_i, v_j\right\rangle\right\} \subseteq \mathscr{E} \wedge \mathbf{mark}(v_j) \neq 0 \Rightarrow \mathscr{E} \coloneqq \mathscr{E} \cup \left\{\left\langle v_j, v_k\right\rangle\right\} \quad (\forall\, i, j, k \in \mathbb{N}^*) \tag{5}$$

Example. We explain how to convert the generalized path $\texttt{alarm(x1)} \leftarrow gr_3^2 \leftarrow \texttt{path(i2, x1)} \leftarrow gr_2^2 \leftarrow \texttt{path(i2, i2)} \rightarrow gr_2^5 \rightarrow \texttt{path(i2, y)} \rightarrow gr_2^6 \rightarrow \texttt{path(i2, x2)} \rightarrow gr_3^5 \rightarrow \texttt{alarm(x2)}$ in Figure 6 to the path from $\texttt{alarm(x1)}$ to $\texttt{alarm(x2)}$ in Figure 7. Here $v_0 = \texttt{alarm(x1)}$ and only $\mathbf{mark}(v_0) = 1$ while $\mathbf{mark}(v) = 0$ for any other variable v. By using rule (1), we construct $\left\langle \texttt{alarm(x1)}, gr_3^2\right\rangle$ in $\mathcal{G}$. Then by applying rule (3) three times, we construct $\left\langle gr_3^2, \texttt{path(i2, x1)}\right\rangle$, $\left\langle \texttt{path(i2, x1)}, gr_2^2\right\rangle$ and $\left\langle gr_2^2, \texttt{path(i2, i2)}\right\rangle$. After that, $\left\langle \texttt{path(i2, i2)}, gr_5^2\right\rangle$ is constructed using rule (4). The final directed path from gr_5^2 to $\texttt{alarm(x2)}$ is constructed using rule (2) five times.

The calculation of those translation rules can be seen as repeatedly applying the information-transmission rules about three nodes X, Y, Z shown previously. We also apply the cycle elimination algorithm in Section 3.3 to $\mathcal{G}$, which removes redundant information transmission. For a directed path in $\mathcal{G}$, we define its corresponding generalized path as any generalized path that share the same node sequence. The theorem states constructed graph $\mathcal{G}$ satisfies completeness and soundness:

Theorem 5.1 (Completeness and Soundness). *Suppose the currently observed alarm node q_1 and another node v_{end} are not D-separated by the set $Q_{observed}$, a directed path from q_1 to v_{end} exists in $\mathcal{G}$ if and only if its corresponding generalized path in $G(\mathcal{D}, A)$ exists and is not blocked by $Q_{observed}$.*

Based on Theorem 5.1, removing or preserving the connection from q_1 to v_{end} on $\mathcal{G}$ can be regarded as alleviating or protecting the generalization between corresponding alarms. The reader can refer to **Appendix** for the proof of Theorem 5.1.

5.3 Refining Abstractions via Max-SAT

This section describes how to refine abstractions to preserve desired generalizations and remove undesired generalizations using $\mathcal{G}$. Such desired/undesired generalizations are identified by whether

probabilities of sampled alarms in $\bar{q}$ are changed correctly after adding new evidence. For a true alarm, the change is correct if and only if its confidence value is higher, and for a false alarm the direction should be the opposite. We refer to the set of alarms whose confidence values change incorrectly as $\bar{q}_F$, and the set whose values change correctly as $\bar{q}_T$. We attempt to separate q_1 from $\bar{q}_F$ while preserving the connection to $\bar{q}_T$ in $\mathcal{G}$. According to Section 3.2, one can find a more precise abstraction $A_{refined}$ which differs from the current abstraction A by changing only in a set $A_{select} \subseteq A$. We approximate the refinement effect by removing all tuples in A_{select}, which also removes all tuples that cannot be derived in Datalog after removing A_{select}. To find the cheapest refined abstraction, we aim to find the smallest A_{select}. For convenience, we define $A_{remain} = A \setminus A_{select}$ and the problem becomes finding the largest A_{remain}. The problem is defined as below:

Definition 5.2 (The Abstraction Refinement Problem). For a derivation graph $G(\mathcal{D}, A) = [V, E]$ and the corresponding information-transmission graph $\mathcal{G}[V, \mathcal{E}]$, let $[V_{remain}, E_{remain}] = G(\mathcal{D}, A_{remain})$ and $\mathcal{G}_{remain}$ be the induced subgraph $\mathcal{G}[V_{remain}]$ for any A_{remain}, the abstraction refinement problem is to find the largest $A_{remain} \subseteq A$ such that on $\mathcal{G}_{remain}$ there is no directed path from q_1 to any node of $\bar{q}_F$, while for every node of $\bar{q}_T$ such a path starting from q_1 still exists.

The connectivity preserving minimum node cut (CPMNC) problem that seeks a minimum node-cut to separate a pair of source and destination nodes and meanwhile ensures the connectivity between the source and its partner nodes has been studied in a previous work [15]. That work shows that CPMNC is NP-hard, and it even cannot be approximated within $\alpha \log n$ for some constant α unless P=NP. Therefore we reduce the refinement problem to the partial weighted Max-SAT problem [31] and solve it using a Max-SAT solver Open-WBO [32].

A partial weighted Max-SAT formula consists of hard constraints ψ_j $(0 \leq j < m)$ and pairs (ω_k, ψ_k) of weights ω_k and soft constraints ψ_k $(m \leq k \leq n)$. All constraints are in the disjunctive normal form of boolean variables. For every node $v \in V$ which may be a tuple or a ground rule, we use itself to represent the boolean variable of whether it is derivable in the Datalog derivation. We denote $reach(v)$ as the boolean variable of whether there is a directed path from q_1 to v in $\mathcal{G}$, and we denote $reach(V) = \{reach(v) \mid v \in V\}$. Then we define the satisfaction relation $\models$, where we use a Boolean variable set V_M to denote assignments to all Boolean variables such that a variable is set to true if and only if it is included in the set:

Definition 5.3 (Satisfaction of Constraints). For a boolean variable set V_M and a constraint ψ which has the form $\psi = \bigvee_{i=1}^{t} f_i \vee \bigvee_{j=t+1}^{s} \neg f_j, s \geq t \geq 0$. $V_M \models \psi$ if and only if $\exists\, i \in \{1, \ldots, t\}, f_i \in V_M$ or $\exists\, j \in \{t+1, \ldots, s\}, f_j \notin V_M$. For a set $\Psi = \{\psi_i \mid i = 1, \cdots, m\}$, $V_M \models \Psi$ if and only if $V_M \models \psi_i$ $(\forall\, i = 1, \cdots, m)$. If $V_M \models \Psi$, we say that V_M is a **model** of Ψ.

We denote the set of hard constraints as Ψ_{hard}. The solution to our Max-SAT problem is a **model** of Ψ_{hard} such that the weight sum of satisfied soft constraints is maximized, which is formulated as:

$$\begin{array}{llll} \textit{Find } V_M \subseteq V \cup reach(V) & \textit{that} & \textit{maximizes} & \Sigma\,\{\omega_i \mid V_M \models \psi_i \textit{ and } m \leq i \leq n\} \\ & & \textit{subject to} & V_M \models \psi_j \textit{ for } 0 \leq j < m \end{array}$$

Due to space concerns, we formally describe our Max-SAT encoding and prove its correctness in the **Appendix**. Briefly, the hard constraints encode the information-transmission relation from our graph and the deriving relation from Datalog inference, and the goal to cut the information transmission that leads to undesired generalization while keeping the transmission that leads to desired generalization. The soft constraints encode the objective to refine as few abstraction tuples as possible. We use an example to walk through the encoding below:

Example. The example is from Section 2 and the encoded formula has two parts. The first part is encoding the derivation in $G(\mathcal{D}, A)$ which is partly shown in Figure 6:

- Abstractions are encoded as soft constraints with weight 1: For instance, $(1, \texttt{abs}(e_1^0))$. This encodes the objective to keep as many abstraction tuples as possible.
- Other input tuples such as $\texttt{path(i2,i2)}$ and $\texttt{taint(i2)}$ are hard constraints.
- For the ground rule gr_3^2 in Figure 6. Its forward derivation is encoded as: $\neg\,\texttt{taint(i2)} \vee \neg\,\texttt{path(i2,x1)} \vee gr_3^2$. Suppose gr_3^1 and gr_3^2 both can derive $\texttt{alarm(x1)}$, they are encoded as: $\neg\, gr_3^1 \vee \texttt{alarm(x1)}$ and $\neg\, gr_3^2 \vee \texttt{alarm(x1)}$.
- As for the backward derivation, we have $\neg\,\texttt{alarm(x1)} \vee gr_3^1 \vee gr_3^2$. Because if $\texttt{alarm(x1)}$ is derivable, there must be a derivable ground rule to derive it. We also have $\neg\, gr_3^2 \vee \texttt{path(i2,x1)}$ and $\neg\, gr_3^2 \vee \texttt{taint(i2)}$. Because if a ground rule is derivable, all its body tuples are derivable.

The other part encodes connectivity on the information-transmission graph $\mathcal{G}[V, \mathcal{E}]$ in Figure 7:

- In Figure 7, we need to separate $\texttt{alarm(x1)}$ from $\texttt{alarm(x2)}$ while keep connections to $\texttt{alarm(x3)}$, and it is encoded as: $\mathit{reach}(\texttt{alarm(x1)})$, $\neg\,\mathit{reach}(\texttt{alarm(x2)})$ and $\mathit{reach}(\texttt{alarm(x3)})$.
- For an edge $\langle u, v\rangle$ in $\mathcal{G}$, if $\mathit{reach}(u)$ is true and v is derivable, we can get $\mathit{reach}(v)$. For example, we have the encoding: $\neg\,\mathit{reach}(gr_3^3) \vee \neg\,\texttt{alarm(x3)} \vee \mathit{reach}(\texttt{alarm(x3)})$.
- If $\mathit{reach}(w)$ is true, w must be derivable, which is encoded as: $\neg\,\mathit{reach}(w) \vee w$. Moreover, there must exist a node connecting to w can be reached from q_1. Take the $\texttt{alarm(x3)}$ in Figure 7 for instance, the encoding is: $\neg\,\mathit{reach}(\texttt{alarm(x3)}) \vee \mathit{reach}(gr_3^3) \vee \mathit{reach}(gr_3^4)$.

The MaxSAT problem of Section 2 is satisfiable and solving it yields $A_{remain} = \{\texttt{abs}(e_1^0), \texttt{abs}(e_3^0), \texttt{abs}(e_5^0), \texttt{abs}(f_1^0), \texttt{abs}(f_3^0), \texttt{abs}(f_5^0)\}$ and $A_{select} = \{\texttt{abs}(g^0)\}$. By changing $\texttt{abs}(g^0)$ to $\texttt{abs}(g^1)$, we get $A_{refine} = \{\texttt{abs}(e_1^0), \texttt{abs}(e_3^0), \texttt{abs}(e_5^0), \texttt{abs}(f_1^0), \texttt{abs}(f_3^0), \texttt{abs}(f_5^0), \texttt{abs}(g^1)\}$.

Note that there exist situations where the abstractions in A_{select} cannot be further refined, and hence $A_{refined} = A$. In such a situation the r in Line 3 of Algorithm 2 will be set to Unsatisfiable, and hence function Refine will return (A, false). In other words, the abstraction will not be refined.

5.4 Application Scope of Our Approach

Finally, we discuss what analyses our approach can apply to. As mentioned in Section 5.2, to ensure refinement can reduce false generalizations, it requires a more precise abstraction will not add correlation between analysis facts (**Assumption 1**). As more precise abstractions typically lead to less confusion between program states, this assumption usually holds. In addition, a sufficient condition for **Assumption 1** is that if abstraction A is more precise than abstraction B ($B \sqsubseteq A$), the abstract states derived by B should overapproximate those of A. Many popular parametric analyses satisfy this condition, such as different k-limited pointer analyses and the predicate abstraction.

Moreover, it is possible to expand our approach beyond analyses expressed in Datalog. In fact, besides the above **Assumption 1**, our approach only requires the analysis derivation can be encoded as a graph, in order to cast the refinement problem as a graph cut problem and perform efficient probabilistic inference. We refer to this graph encoding requirement as **Assumption 2**. In theory, the derivation of any abstract-interpretation-based analysis can be converted into a graph. In the most general sense, an abstract state can be a node, and a transition from one abstract state to another can be an edge in the graph. In practice, there may exist more efficient encodings for individual analyses. Previously, we focused mainly on the analyses that are expressed in Datalog and its variants because they cover a large class of important analyses and BayesRefine can automate the graph construction process for them. For analyses that cannot be expressed in Datalog, whether BayesRefine can efficiently handle them varies case by case. In the **Appendix**, we discuss how to apply our approach to an interval analysis as an example.

6 Empirical Evaluation

Our evaluation seeks to answer the following questions:

Q1. How effective is BayesRefine at reducing user interactions for Bayesian program analysis?
Q2. How effective is BayesRefine at complementing BinGraph when its training set selection is sub-optimal?
Q3. How effective is BayesRefine compared with BayeSmith [21], which addresses the over-generalization problem through parameter learning?
Q4. How sensitive is the improvement brought by BayesRefine and how adversely is it affected by changing parameters (k_1 and k_2)?
Q5. Does the abstraction become much more expensive after being refined by BayesRefine?
Q6. How scalable are different components of BayesRefine?

6.1 Experimental Setup

We conducted all our experiments on Linux machines with 2.6 GHz processors and 256 GB RAM running Oracle HotSpot JVM 1.6, LibDAI version 0.3.2 and Open-WBO version 2.1. We use the Chord framework [37] for Datalog derivation and the Bingo framework [39] for Bayesian inference.

Our Approach and Baselines. To demonstrate the effectiveness of BayesRefine, we apply it to refine the abstractions produced by BinGraph [53] (Refine-B). We also apply BayesRefine to refine the coarsest abstractions (Refine-C) to consider the case where BinGraph is not applicable due to issues like lack of training data. As baselines, we consider running Bingo without refinement using those abstractions produced by BinGraph (Base-B), those coarsest abstractions (Base-C), as well as the most precise abstractions (Base-P).

Instance Analyses. Following the setting in BinGraph [53], we conduct our evaluation on two static analyses, a Datarace analysis [37] and a thread-escape analysis [38], whose Datalog implementations comprise 102 rules, 58 input relations, 44 output relations and 60 rules, 34 input relations, 27 output relations respectively. Next we describe them in more detail. (1) **Datarace analysis.** It finds all possible statement pairs which may operate on the same heap object simultaneously with at least one write operation. It combines thread-escape, may-happen-in-parallel, and lock-set analyses that are flow-and-context sensitive. They build upon call-graph and aliasing information obtained from a pointer analysis. The analysis is intended to be sound [28]. (2) **Thread-escape analysis.** It generates queries about thread locality: A heap object in a multi-threaded program is thread-local when it is accessible only from at most a single thread. The thread-escape analysis used in our evaluation is flow-insensitive and context-insensitive, which differs from the datarace analysis.

The abstraction A we use to parameterize either analysis specifies precision parameters independently for each object allocation site h in the program. With a slight abuse, we consider A as a map from allocation sites to their precision parameters, and those abstractions are ordered as: $A_1 \sqsubseteq A_2 \Leftrightarrow \forall h : A_1(h) \leq A_2(h)$. For each h, an input tuple $\texttt{ABS}(h, A(h))$ will be constructed when encoding as Datalog program. (1) For the datarace analysis, its abstraction parameterization comes from the k-object-sensitivity pointer analysis [33]: It distinguishes different calling contexts of methods by defining them as allocation sites of receiver objects, and it associates a string of such allocation sites of length upto k. A different k value can be associated with each allocation site h, and $A(h)$ is set to be k since the abstraction precision depends heavily on how many distinct calling contexts it considers. For scalability, we set the maximum value for each k to be 3. (2) For the thread-escape analysis, it is parameterized by the heap abstraction. An allocation site h is reasoned alone (denoted as $A(h) = 1$) or it is merged into a global heap abstraction (denoted as $A(h) = 0$) for efficiency. In the coarsest setting, all objects in all allocation sites will be considered as one object.

Ground truth. We get the ground truth for each alarm (whether it is true) beforehand. We simulate the whole interactive alarm interaction automatically to calculate relevant metrics. For the datarace analysis, the ground truth comes from Bingo's artifact. While for the thread-escape analysis, we used a CEGAR-based flow- and context-sensitive analysis [51] to obtain ground truth. The analysis

Table 1. Benchmark characteristics. 'Total' and 'App' columns are numbers using 0-CFA call graph construction, with and without the JDK for all the benchmarks.

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

is highly precise but also too expensive to apply broadly in practice. Such a setup allows to study how our approach helps address one major source of imprecision of program analysis – the tradeoff between precision and efficiency. With the help of oracle feedback, the Bayesian approach can improve the precision of an analysis without degrading its efficiency significantly.

Benchmarks. We use a suite of 12 Java programs shown in Table 1 following BinGraph [53], including programs commonly used in past work from the Ashes Suite [16, 49] and the DaCapo suite [5]. In the evaluation of BinGraph [53], six benchmarks are used for the datarace analysis and six are used for the thread-escape analysis, but we exclude `xalan` for the datarace analysis since BinGraph ran out of its memory limit (40GB) on `xalan` as it finds overly expensive abstractions.

Parameter settings. For parameter k_1, we set it to 0 initially. In order to accelerate our approach, when the refinement module fails to perform the abstraction refinement (e.g. the change direction of the sampled alarms' probability all match their ground truth), BayesRefine will increase k_1 by a fixed step size *step*, in other words, adjust k_1 to k_1 + *step*. This is to prevent excessive sampling that cannot perform abstraction refinement, which will bring more workload to users. By conducting experiments on the training benchmark `weblech`, we find the best is to set the sampled number k_2 to 2 for all benchmarks, while setting *step* to be 2 for all benchmarks in the thread-escape analysis, and in the datarace analysis setting it to be: $\lceil \frac{|Q \cap [[\mathcal{D}, A_0]]|}{700} \rceil \times 2$. Here A_0 is the initial abstraction, and $|Q \cap [[\mathcal{D}, A_0]]|$ is the number of output alarms using A_0.

Metrics. The main metric we use is Rank-100%-FP, which is the total number of inspected false alarms when all the true alarms have been discovered. It reflects the user's burden in the interaction. We also use another similar metric Rank-90%-FP, which is the total number of observed false alarms when 90% of the true alarms are discovered. When calculating these two metrics, we have included the additionally inspected alarms posed by BayesRefine. Compared to the original metrics Rank-100%-T and Rank-90%-T used in previous work [39, 53], which count the number of all observed alarms, we choose to not count the number of inspected true alarms. This is because there is a large number of true alarms using the thread-escape analysis, as shown in Table 1. As for the inversion count used in BinGraph [53], since the output alarms $Q \cap [[\mathcal{D}, A]]$ will change with the abstraction A changing when applying BayesRefine, it is hard to compare this metric and we do not use it.

6.2 Effectiveness of Reducing Interactions

To demonstrate the effectiveness of BayesRefine, we present statistics of our two metrics in Table 2. We discuss Rank-100%-FP first. Over all benchmarks, for Refine-B, the user needs to inspect 18.86% fewer false alarms than Base-B for datarace on average, and 33.01% fewer for thread-escape. For example, the datarace analysis produces 187 alarms on `raytracer` using the abstraction produced by

Table 2. Summary of metrics for effectiveness of BayesRefine. "Average Reduction" shows the average reduction ratio using BayesRefine compared to corresponding baselines Base-B and Base-C on column Refine-B and Refine-C respectively. On column Base-P, it shows the reduction from Base-P to Refine-B. Baselines Base-B, Base-C and Base-P refer to running Bingo without refinement using abstractions produced by BinGraph, the coarsest abstractions and the most precise abstractions respectively. Columns # **Alarms** present the number of alarms using the initial abstractions.

	Program	# Alarms			Rank-100%-FP					Rank-90%-FP				
		Base-C	Base-B	Bugs	Refine-B	Base-B	Refine-C	Base-C	Base-P	Refine-B	Base-B	Refine-C	Base-C	Base-P
Datarace analysis	avrora	1230	1000	29	**681**	732	711	909	688	**114**	338	220	395	366
	ftp	571	525	75	9	9	28	94	**8**	9	9	28	44	**7**
	sunflow	2288	1561	171	**185**	188	241	289	*failed*	**82**	103	106	275	*failed*
	raytracer	187	187	3	**1**	7	30	29	6	**1**	7	30	29	6
	luindex	1,236	976	2	**11**	11	16	16	323	**11**	11	16	16	323
Average Reduction					18.86%↓		21.03%↓		42.11%↓	34.47%↓		43.60%↓		55.05%↓
Thread-escape analysis	hedc	552	380	287	**51**	85	60	109	92	**36**	45	53	79	58
	jspider	754	645	430	**136**	205	180	232	188	**51**	64	119	113	140
	montecarlo	353	89	54	**17**	24	51	109	24	**7**	10	49	107	12
	pool	497	432	312	**47**	59	80	90	83	**42**	**42**	79	88	80
	raytracer	422	319	233	**45**	54	66	81	86	**28**	53	57	77	76
	toba-s	1325	1278	998	**117**	280	158	224	280	65	**62**	157	154	251
Average Reduction					33.01%↓		29.95%↓		41.78%↓	18.77%↓		19.34%↓		54.66%↓

BinGraph, of which 3 are real races. BayesRefine identifies all true alarms with the user inspecting only 1 false alarm, compared to 7 for Base-B. Another notable example is `toba-s`, on which the thread-escape analysis produces 1278 alarms. Of these alarms, 998 are true alarms, and BayesRefine reports them while presenting just 117 false alarms. On the other hand, Base-B presents 280 false alarms, which include all the false alarms. To conclude, BayesRefine substantially reduces the user's workload - the wasteful effort on inspecting false alarms, and Refine-B outperforms Base-B for 9 out of 11 benchmarks. For the remaining programs `ftp` and `luindex` Refine-B and Base-B perform the same. This is mainly because BinGraph has already found good initial abstractions.

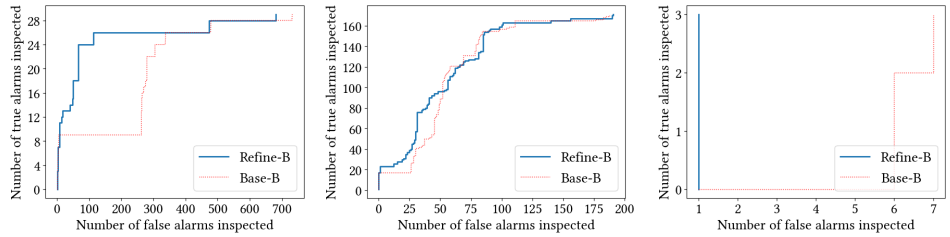
When starting with the coarsest abstraction, Refine-C outperforms Base-C on 9 of 11 benchmarks, with average reduction ratios of 21.03% and 29.95% for the two analyses respectively. This shows that BayesRefine can also reduce inspection burden when an offline abstraction selection method like BinGraph is not applied. Additionally, Refine-B outperforms Base-P with average reduction ratios of 42.11% and 41.78% for the two analyses, which shows that the abstractions refined by BayesRefine can have better generalization ability than the most precise abstractions.

As for Rank-90%-FP, Refine-B outperforms Base-B with average reduction ratios of 34.47% and 18.77% for the two analyses respectively, and Refine-C outperforms Base-C with average reduction ratios of 43.60% and 19.34% respectively. The increase in average reduction ratios compared to Rank-100%-FP on datarace is mainly because the lastly presented few true alarms are sometimes outliers that do not correlate with previously inspected alarms and thus prolong the entire interaction process, which is evident for both `avrora` and `sunflow`. On the other hand, the decrease for thread-escape is because most of the alarms are true, and BayesRefine does not refine the abstractions often in the early stage as false generalization rarely happens.

In practice, the user would not know when all true alarms have been found but may be willing to inspect a certain number of alarms instead. In order to measure the performance of BayesRefine in this scenario, we assume that the user only inspects 20% of the initially generated alarms, and show the numbers of observed true alarms in Table 3. BayesRefine enables the user to inspect 20.52% more true alarms than BinGraph on average, which demonstrates its practicality.

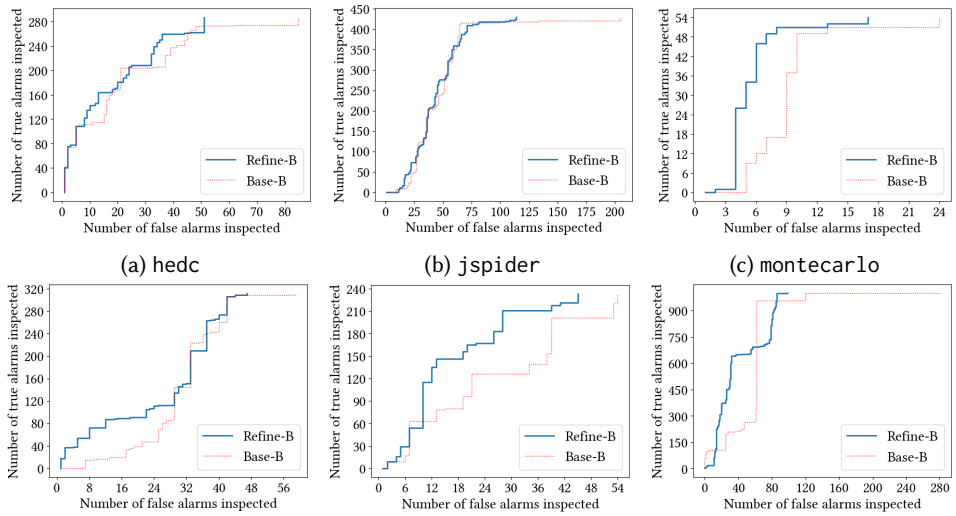
Table 3. Summary of the numbers of observed true alarms when we assume the user only inspects 20% of the initially generated alarms.

Setting	Datarace analysis					Thread-escape analysis					
	avrora	ftp	sunflow	luindex	raytracer	hedc	jspider	montecarlo	pool	raytracer	toba-s
Base-B	9	9	165	2	3	74	101	12	61	57	213
Refine-B(20.52%↑)	26	9	159	2	3	74	110	14	71	54	221



(a) avrora (b) sunflow (c) raytracer

Fig. 13. The ROC curves for the datarace analysis.



(a) hedc (b) jspider (c) montecarlo

(d) pool (e) raytracer (f) toba-s

Fig. 14. The ROC curves for the thread-escape analysis.

In addition, to capture the dynamic behavior of the interaction process, we plot ROC curves [17] for the datarace analysis in Figure 13 and for the thread-escape analysis in Figure 14, which is also presented in previous research [39, 53]. When the user has inspected x false alarms and y true alarms, a point (x, y) is plotted in the ROC curve. Note that we exclude `ftp` and `luindex` as their underlying abstractions are not refined by BayesRefine and the ROC curves are the same for Base-B and Refine-B. To conclude, BayesRefine outperforms BinGraph not just in the aggregate, but in most of the individual interactions.

Table 4. Summary of metrics under different training sets in the datarace analysis. `ftp-jspider` represents using {`ftp`, `jspider`} for training in BinGraph to produce initial abstractions. `ftp-hedc` represents using {`ftp`, `hedc`}. Column **Refine-B** and **Base-B** present corresponding Rank-100%-FP running BayesRefine and running Bingo without refinement respectively. Column # **Alarms** presents the number of alarms using the initial abstractions.

	Program	ftp-jspider			ftp-hedc		
		# Alarms	Refine-B	Base-B	# Alarms	Refine-B	Base-B
Datarace analysis	avrora	982	688	631	989	564	534
	sunflow	1,540	279	1,457	1,540	223	1,369
	raytracer	187	1	30	187	9	11
	luindex	1,066	17	21	1053	299	299
Average Reduction			46.88%↓			24.07%↓	

6.3 Effectiveness in Improving BinGraph under Different Training Sets

As mentioned earlier, the performance of BinGraph may vary depending on the training programs selection. In this section, we evaluate how effective BayesRefine is in improving BinGraph under different training sets, especially when the training set selection is sub-optimal.

For the datarace analysis, the original BinGraph uses { `jspider`, `hedc` } for training as they are relatively small and provide sufficient number of alarms. We additionally consider `ftp` which is comparable in these two metrics, and get two new training sets {`ftp`, `jspider` } and {`ftp`, `hedc`}. The other programs are too large, making training unscalable. Table 4 summarizes the results. The average reduction ratio in user interaction further improves to 46.88% and 24.07% respectively as the training set selection is less optimal for most benchmarks. In particular, the number of inspected false alarms using BinGraph (Base-B) on `sunflow` increases drastically, from 188 to 1,457 and 1,369 respectively, which are even much larger than that of directly applying Bingo with the coarsest abstraction (Base-C)! Our approach addresses this issue and reduces the numbers to 279 and 223, which are also better than the original Bingo. We see similar results on `raytracer` when the training set is { `ftp`, `jspider` }. As for `avrora`, Refine-B yields comparable results as Base-B, because the results of Base-B further improve on `avrora` with the new training sets so there is less room for our approach to improve it. One outlier is the result on `luindex` with {`ftp`, `hedc` } as the training set. In this setting, the performance of Base-B degrades severely and our approach also fails to salvage it. This is because there are only two true alarms for `luindex` which are discovered as the last two alarms out of the 299 alarms. Before these two alarms, our approach only samples false alarms and fails to refine any abstraction, and thus degrades to Base-B. Overall, the improvement of our approach on Bingo is even more significant when the training set selection is less ideal.

For thread-escape, the original BinGraph paper [53] conducted a leave-one-out cross-validation experiment and showed that BinGraph is less sensitive to the training set selection for thread-escape. We apply our approach to start with the abstractions produced from the experiment. As Table 5 shows, BayesRefine improves BinGraph with an average reduction ratio of 20.61%.

6.4 Effectiveness in Comparison with BayeSmith

To reduce the impact of over-generalization, BayeSmith [21] proposes a more fine-grained way to assign probabilities by attaching syntactical predicates to rules. In machine learning terms, it falls into the category of parameter learning while our approach is structure learning, and these two approaches are complementary and address the generalization problem from two different angles. We conduct an empirical comparison to show their relation.

Table 5. Summary of metrics under leave-one-out cross-validation in the thread-escape analysis. Column **Base-B**, **Refine-B**, **Base-B+S** and **Refine-B+S** all present corresponding Rank-100%-FP. Column **Base-B** and **Base-B+S** present running Bingo without refinement using the default probabilities in BinGraph and probabilities learnt by BayeSmith respectively, while **Refine-B** and **Refine-B+S** present results of BayesRefine. Column # **Alarms** presents the number of alarms using the initial abstractions

	Program	# Alarms	Base-B	Refine-B	Base-B+S	Refine-B+S
Thread-escape analysis	hedc	380	55	55	53	53
	jspider	678	232	145	154	136
	montecarlo	89	11	11	8	8
	pool	432	59	47	61	60
	raytracer	319	86	64	57	41
	toba-s	1278	280	154	280	144
	weblech	512	126	106	123	81
Average Reduction				20.61%↓	13.89%↓	29.60%↓

Table 6. Sensitivity of BayesRefine to varying parameters of the threshold k_1 for continuously observed false alarms and the sampling number k_2 and to user mistakes using the thread-escape analysis.

	Program	Base-B	K-(2, 1)	K-(1, 2)	K-(2, 2)	K-(3, 2)	K-(2, 3)	Base-N	Refine-N
Thread-escape analysis	hedc	85	58	51	51	60	52	99.8	79.0
	jspider	205	137	127	136	137	130	218.7	121.3
	montecarlo	24	16	15	17	17	11	28.2	15.9
	pool	59	47	75	47	47	45	88.2	72.1
	raytracer	54	45	45	45	48	68	67.5	61.7
	toba-s	280	142	101	117	146	137	257.5	143.3
Average Reduction			30.76%↓	28.17%↓	33.01%↓	28.51%↓	29.74%↓		30.03%↓

Following the learning setting of BayeSmith [21], we conduct a leave-one-out cross-validation for thread-escape: We use the same train/test setting as the previous leave-one-out cross-validation experiment for thread-escape inherited from BinGraph [53]. Hence, in each fold, BinGraph uses the same training set as Bayesmisth to produce initial abstractions for test programs. As a baseline, we run Bingo without refinement using probabilities learnt by BayeSmith (Base-B+S). Then we apply BayesRefine to refine abstractions while using the learned probabilities (Refine-B+S).

The results are appended to Table 5, where Refine-B+S and Base-B+S outperform Base-B with an average reduction ratio of 29.60% and 13.89% respectively. Note the reduction of applying BayeSmith alone is less than applying BayesRefine alone (Refine-B). This shows that by refining on the fly, BayesRefine can resolve over-generalization in a more fine-grained manner, and more importantly, there are cases where changing parameters without changing the structure is not enough to resolve over-generalization. Finally, combining BayesSmith and BayesRefine produces the best results, which indicates that the two approaches are complementary, and BayesRefine can still improve the abstractions using rule probabilities learnt by BayesSmith.

6.5 Sensitivity to Values of Parameters k_1 and k_2 and to User Mistakes in BayesRefine

In order to measure how k_1, the threshold of continuously observed false alarms, and k_2, the number of sampled alarms per iteration, affect our whole framework, we evaluate the performance of BayesRefine on thread-escape under different values of them. As mentioned before, k_1 increases gradually on the fly and is controlled by another parameter *step*. So we study the effect of using different $(step, k_2)$ instead. We fix one parameter to the default value used in previous experiments (i.e., 2) and change the other to its nearest natural numbers, and measure the Rank-100%-FP metrics.

Table 7. Efficiency of the refined datarace analyses including the size of the refined abstraction, the running time of the refined analyses and the number of alarms before and after refinement. Since in `luindex` and `ftp` the abstraction is not refined, they are not included.

Program	Abstraction Size				Time for Datalog Analysis(s)				#Alarms			
	Base-B	Final	Max	Sparsity	Base-B	Final	%Increased	Max	Base-B	Final	%Reduced	Max
avrora	1153	3836	15837	50.75%	237.9	275.5	15.80%	634.8	1000	978	2.20%	978
sunflow	1236	4943	16386	47.31%	491.5	570.4	16.05%	10,684.3	1561	1027	34.21%	958
raytracer	26	36	4050	2.07%	26.9	28.1	4.65%	29.3	187	187	0%	187

Table 6 shows the results, where $\mathbf{K}-(p_1,p_2)$ represents the results of Refine-B when using $step = p_1$ and $k_2 = p_2$. As expected, the default setting $\mathbf{K}-(2,2)$ performs the best as it is chosen via a training process, but Refine-B still outperforms Base-B with average reduction ratios over 28% on Rank-100%-FP for other settings. To conclude, the results show BayesRefine can robustly tolerate reasonable changes of its parameters.

We also measure the sensitivity of BayesRefine to incorrect user answers by injecting random noise: for each initially generated alarm, we flip its ground truth with probability 5%. As a baseline, we run BinGraph without refinement in this noisy setting (Base-N). Then we apply BayesRefine to refine abstractions (Refine-N). We run the experiment in each setting 30 times, calculate the averages of all statistics and append the results to Table 6. Since Refine-N outperforms Base-N with an average reduction ratio of 30.03%, BayesRefine is resilient to user mistakes.

6.6 Experimental Results about the Efficiency of the Refined Analyses

In order to measure the efficiency of analyses refined by BayesRefine, we record the running time of the Datalog-based analyses using the final refined abstractions. Since the running cost of the Datalog-based thread-escape analyses is negligible for all the benchmarks (less than 15s), we only present statistics of the datarace analysis. In addition, we compute the abstraction size, which is also an estimate of how costly it is to run the analysis. An abstraction is considered to be more efficient when its size is smaller, and the *size of abstraction* A is defined the same for the two analyses: For both datarace analysis and thread-escape analysis, it is $\Sigma_h A(h)$.

Table 7 summarizes results about efficiency of refined datarace analyses. The **Max** columns show the results using the largest abstraction in the family for each program. For the datarace analysis, it corresponds to using a 3-objective-sensitive pointer analysis. The **Final** columns show the results using the abstraction produced by the last refinement, while the **Base-B** columns show results using the initial abstraction produced by BinGraph. The **Sparsity** column shows the percentage of abstraction parameters ($A(h)$) that are not 0. In all cases, the size of the final abstraction produced by our approach is less than $\frac{1}{3}$ of the maximum size, while the largest sparsity is only 50.75%. It shows that the abstraction is not refined too much.

As for the time of running Datalog-based datarace analyses, the increase ratios from **Base-B** to **Final** are 15.80%, 16.05%, 4.65% for `avrora`, `sunflow` and `raytracer` respectively. The final running time is also much less than that using the max-sized abstraction for all benchmarks. For `avrora`, one of our largest benchmarks, the final running time is less than 50% of that of the **Max**.

At last, columns '#**Alarms**' compare the number of alarms derived by the Datalog-based analyses using initial, finally refined and max-sized abstractions. An interesting discovery is that on `raytracer` and `avrora` the Datalog-based datarace analyses using final abstractions derive the same or almost the same alarms as they do using initial abstractions, while in those benchmarks BayesRefine successfully reduces the number of interactions. In fact, even using the most precise analysis based on the 3-objective-sensitive pointer analysis, we cannot decrease the number of derived

Table 8. Efficiency of different components of BayesRefine on datarace. The numbers of tuples and relevant ground clauses are counted from the derivation graphs produced by the final refinement. The time costs are average values in seconds per iteration during the interaction.

Program	# Tuples		# Relevant ground clauses		Inference time (s)		Refinement time (s)
	BayesRefine	Base-B	BayesRefine	Base-B	BayesRefine	Base-B	BayesRefine
avrora	6,408	7,339	10,540	12,478	51.8	67	0.35
sunflow	18,243	18,241	29,558	29,535	260.8	295	19.1
raytracer	1,089	1595	1,387	1296	0.54	0.59	0.06

alarms on these 2 benchmarks. This shows that BayesRefine can reduce user's workload through suppressing the over-generalization in scenarios where increasing abstraction precision makes little improvement for traditional program analyses.

6.7 Scalability

Table 8 shows the average refinement time and the average Bayesian inference time both per iteration. We exclude `ftp` and `luindex` as their underlying abstractions are not refined by BayesRefine. First, we observe BayesRefine itself incurs little overhead as the refinement takes less than 5% of the time consumed by running Datalog and Bayesian inference combined. Second, the maximum growth of the final Datalog analysis time is only 16.05% after applying BayesRefine corresponding to Table 7. Third, the average Bayesian inference time actually decreases after applying BayesRefine, as more precise abstractions lead to smaller Bayesian networks in many cases. As a validation, Table 8 shows that after applying BayesRefine, the numbers of tuples and relevant ground clauses both decrease. We observe similar results for thread-escape.

7 Related Work

Bayesian Program Analysis. Bayesian analysis resolves alarms and improves accuracy by leveraging posterior information, such as the users' feedback [39, 49], the old version of the program [19], and dynamic analysis results [11]. The main idea behind such analysis is to compute the confidence value for each alarm and updates rankings based on received posterior information.

Most existing works focus on how to integrate different posterior information except BayeSmith [21] and BinGraph [53], which also try to address over-generalization. The former tackles the problem by proposing a more fine-grained way to assign probabilities in Bayesian networks, while the latter applies a data-driven approach to learn an abstraction selection policy from training programs. Concretely, BayeSmith splits a rule in Datalog into multiple ones by attaching syntactical predicates. This does not change the structure of the corresponding Bayesian network on a program but gives more flexibility in assigning the conditional probabilities as the edges ground from the same Datalog rule have the same probability. BinGraph is closer to our approach as it also tries to address over-generalization through abstraction selection. However, it is an offline-learning-based approach and can lead to sub-optimal performance due to its learning nature or the test program being out-of-distribution. BayesRefine addresses this issue by refining abstractions produced by BinGraph. Both approaches are complementary to ours and can be combined to yield better results.

Counterexample-Guided Abstraction Refinement(CEGAR). CEGAR was originally proposed for hardware and software model checking [2, 9, 9, 14]. Later, it is applied in parametric program analysis [18, 50, 51]. As the previous work shows [53], these methods for conventional analyses does not work for our problem. Moreover, although CEGAR techniques have been proposed for finite state probabilistic systems [1, 8], to the best of our knowledge, BayesRefine is the first CEGAR algorithm for models that combine both probabilities and program semantics.

Deep learning based approaches for program analysis. Numerous previous techniques incorporate deep learning to filter false alarms or to detect bugs directly. The former [22, 23, 25] are limited to specific types of analysis and evaluated on synthetic benchmarks or manually collected small-scale data sets (such as the Juliet benchmark [6]). It is unclear how to engineer the models for complex analyses like the datarace analysis and whether they would perform well on real-world benchmarks like Dacapo. On the other hand, our approach is more general and requires little manual effort to instantiate in an analysis. As for neural models including GNNs [12, 13, 46] that detect bugs directly, similarly, they are typically carefully tailored to specific analyses, and a study in 2021 [10] shows that their precisions drop drastically on real-world datasets. Moreover, they are more suitable for detecting bugs that can be identified through local patterns such as variable/API misuses, rather than bugs that need complex logical reasoning such as the datarace.

LLM-based approaches for program analysis. Several recent papers have applied LLMs to suppress false alarms reported by a static analyzer [26, 27, 45]. However, these approaches apply highly-customized prompts that are specified to certain analyses and offer no general solution. In addition, LLMs can replace the role of human users in the interaction of our framework, which is complementary to BayesRefine. One can also combine LLMs with our approach to reduce the number of invocations to LLMs.

8 Conclusion

We present BayesRefine, an iterative refinement approach to generate abstractions of better generalization for Bayesian program analysis. It addresses the problem of reducing over-generalization by casting it as a graph cut problem with connectivity by leveraging the theory of conditional independence, which is then solved using a Max-SAT solver. Our evaluation on two representiative analyses shows that our approach significantly improves over the existing approach that generates abstractions beforehand using policies learned offline.

Data-Availability Statement

An artifact that provides the implementation of BayesRefine is available [48]. It includes all the source code, scripts, data, and statistics in our experiments. All results in our experiments can be reproduced.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback and comments. This research was partly supported by the National Natural Science Foundation of China under Grant No. 62172017 and Grant No. W2411051.

References

[1] Husain Aljazzar, Matthias Kuntz, Florian Leitner-Fischer, and Stefan Leue. 2010. Directed and heuristic counterexample generation for probabilistic model checking: a comparative evaluation. In *Proceedings of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*. 25–32.

[2] Thomas Ball and Sriram K. Rajamani. 2002. The SLAM project: debugging system software via static analysis. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, John Launchbury and John C. Mitchell (Eds.). ACM, 1–3. doi:10.1145/503272.503274

[3] Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-Based Static Analysis (Extended Version). arXiv:2009.08361 [cs.PL] https://arxiv.org/abs/2009.08361

[4] Christopher M Bishop and Nasser M Nasrabadi. 2006. *Pattern recognition and machine learning*. Vol. 4. Springer.

[5] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking

development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 169–190.
[6] Tim Boland and Paul E Black. 2012. Juliet 1. 1 C/C++ and java test suite. *Computer* 45, 10 (2012), 88–90.
[7] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.* 44, 10 (oct 2009), 243–262. doi:10.1145/1639949.1640108
[8] Rohit Chadha and Mahesh Viswanathan. 2010. A counterexample-guided abstraction-refinement framework for Markov decision processes. *ACM Transactions on Computational Logic (TOCL)* 12, 1 (2010), 1–49.
[9] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. 2004. Modular Verification of Software Components in C. *IEEE Trans. Software Eng.* 30, 6 (2004), 388–402. doi:10.1109/TSE.2004.22
[10] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering* 48, 9 (2021), 3280–3296.
[11] Tianyi Chen, Kihong Heo, and Mukund Raghothaman. 2021. Boosting static analysis accuracy with instrumented test executions. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1154–1165.
[12] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–33.
[13] Zhaoyang Chu, Yao Wan, Qian Li, Yang Wu, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2024. Graph neural networks for vulnerability detection: A counterfactual explanation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 389–401.
[14] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*. Springer, 154–169.
[15] Qi Duan and Jinhui Xu. 2014. On the connectivity preserving minimum cut problem. *J. Comput. Syst. Sci.* 80 (2014), 837–848.
[16] Mahdi Eslamimehr and Jens Palsberg. 2014. Race directed scheduling of concurrent programs. In *Proceedings of the 19th Symposium on Principles and Practice of Parallel Programming*. ACM, 301–314. doi:10.1145/2555243.2555263
[17] Tom Fawcett. 2006. An introduction to ROC analysis. *Pattern recognition letters* 27, 8 (2006), 861–874.
[18] 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. doi:10.1145/2837614.2837663
[19] Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik. 2019. Continuously reasoning about programs using differential bayesian inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 561–575.
[20] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2013). https://api.semanticscholar.org/CorpusID:14109210
[21] Hyunsu Kim, Mukund Raghothaman, and Kihong Heo. 2022. Learning Probabilistic Models for Static Analysis Alarms. (2022).
[22] Ugur Koc, Parsa Saadatpanah, Jeffrey S Foster, and Adam A Porter. 2017. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN international workshop on machine learning and programming languages*. 35–42.
[23] Ugur Koc, Shiyi Wei, Jeffrey S Foster, Marine Carpuat, and Adam A Porter. 2019. An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool. In *2019 12th ieee conference on software testing, validation and verification (icst)*. IEEE, 288–299.
[24] Daphne Koller and Nir Friedman. 2009. *Probabilistic graphical models: principles and techniques*. MIT press.
[25] Seongmin Lee, Shin Hong, Jungbae Yi, Taeksu Kim, Chul-Joo Kim, and Shin Yoo. 2019. Classifying false positive static checker alarms in continuous integration using convolutional neural networks. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 391–401.
[26] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 474–499.
[27] Ziyang Li, Saikat Dutta, and Mayur Naik. 2024. Llm-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2405.17238* (2024).
[28] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhotak, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58 (2015), 44–46. http://cacm.acm.org/magazines/2015/2/182650-in-defense-of-soundiness/abstract

[29] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to flix: a declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 194–208. doi:10.1145/2908080.2908096

[30] 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*. 462–473.

[31] Vasco M. Manquinho, João Marques-Silva, and Jordi Planes. 2009. Algorithms for Weighted Boolean Optimization. *CoRR* abs/0903.0843 (2009). arXiv:0903.0843 http://arxiv.org/abs/0903.0843

[32] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. 2014. Open-WBO: A Modular MaxSAT Solver,. In *SAT*.

[33] Ana Milanova, Atanas Rountev, and Barbara Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14 (01 2005), 1–41. doi:10.1145/1044834.1044835

[34] Joris M. Mooij. 2010. libDAI: A Free and Open Source C++ Library for Discrete Approximate Inference in Graphical Models. *Journal of Machine Learning Research* 11 (Aug. 2010), 2169–2173. http://www.jmlr.org/papers/volume11/mooij10a/mooij10a.pdf

[35] António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. 2013. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints An Int. J.* 18, 4 (2013), 478–534. doi:10.1007/s10601-013-9146-2

[36] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. 1999. Loopy Belief Propagation for Approximate Inference: An Empirical Study. In *Conference on Uncertainty in Artificial Intelligence*. https://api.semanticscholar.org/CorpusID:16462148

[37] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 308–319.

[38] Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. 2012. Abstractions from tests. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 373–386.

[39] 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*. 722–735.

[40] Thomas W Reps. 1995. Demand interprocedural program analysis using logic databases. In *Applications of Logic Databases*. Springer, 163–196.

[41] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society* 74, 2 (1953), 358–366.

[42] Yannis Smaragdakis. 2010. Pick Your Contexts Well : Understanding Object-Sensitivity The Making of a Precise and Scalable Pointer Analysis. https://api.semanticscholar.org/CorpusID:9747942

[43] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29.

[44] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental whole-program analysis in Datalog with lattices. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1–15.

[45] Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. 2024. LLMDFA: Analyzing Dataflow in Code with Large Language Models. *Advances in Neural Information Processing Systems* 37 (2024), 131545–131574.

[46] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning semantic program embeddings with graph interval neural network. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.

[47] John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*. https://api.semanticscholar.org/CorpusID:14810646

[48] Xin Zhang Yuanfeng Shi, Yifan Zhang. 2025. On Abstraction Refinement for Bayesian Program Analysis (Paper Artifact). https://doi.org/10.5281/zenodo.16917600

[49] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. 2017. Effective interactive resolution of static analysis alarms. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.

[50] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in Datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 239–248.

[51] Xin Zhang, M. Naik, and Hongseok Yang. 2013. Finding optimum abstractions in parametric dataflow analysis. *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2013).

[52] Xin Zhang, Xujie Si, and Mayur Naik. 2017. Combining the Logical and the Probabilistic in Program Analysis. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 27–34.

[53] 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. doi:10.1145/3649845

Received 2025-03-26; accepted 2025-08-12