BEER: Interactive Alarm Resolution in Bayesian Program Analysis via Exploration-Exploitation

HAORAN LIN, Peking University, China
ZHENYU YAN, Peking University, China
XIN ZHANG*, Peking University, China

Interactive Bayesian program analysis enhances static analysis by modeling derivations as probabilistic dependencies, enabling ranking alarms by calculated confidences, proposing highly likely alarms for user inspection, and updating confidences with inspection results. Existing interactive approaches adopt a purely greedy, exploitation-only selection strategy that always inspects the highest-confidence alarm. However, such strategies are prone to local optima, leading to redundant inspections and delayed identification of true alarms. We propose BEER (Bayesian Exploration-Exploitation Ranker), a framework that systematically integrates the Exploration-Exploitation trade-off into Bayesian program analysis. BEER leverages structural correlations between alarms—derived from shared root causes in the Bayesian model—to estimate expected information gain and guide exploration. When repeated false alarms indicate model stagnation, BEER selects alarms from minimally explored, highly correlated clusters to accelerate learning. Implemented atop the BINGO framework, BEER achieves up to 32% effectiveness in ranking efficiency over the greedy baseline on datarace, thread-escape, and taint analyses, demonstrating the efficacy of exploration-guided alarm resolution.

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, exploration-exploitation

ACM Reference Format:
Haoran Lin, Zhenyu Yan, and Xin Zhang. 2026. BEER: Interactive Alarm Resolution in Bayesian Program Analysis via Exploration-Exploitation. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 106 (April 2026), 27 pages. https://doi.org/10.1145/3798214

1 Introduction

Bayesian program analysis [Mangal et al. 2015; Raghothaman et al. 2018; Zhang et al. 2017b] is an emerging paradigm that incorporates probabilities into traditional analyses. It models an analysis's logical derivation as a Bayesian network to infer confidence scores of alarms, which in turn can systematically incorporate external information in posterior probabilities to improve the accuracy of such confidence scores. Such external information includes user feedback [Mangal et al. 2015; Raghothaman et al. 2018], test run information [Chen et al. 2021], differences between program versions [Heo et al. 2019], and natural language information in program texts [Li and Zhang 2025].

*Corresponding author.

Authors' Contact Information: Haoran Lin, haoranlin@stu.pku.edu.cn, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China; Zhenyu Yan, zhenyuyan@stu.pku.edu.cn, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China; Xin Zhang, xin@pku.edu.cn, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China.

ACM 2475-1421/2026/4-ART106
https://doi.org/10.1145/3798214

While the external information and the corresponding usage scenario vary, the earliest and the most prominent application of Bayesian program analysis is interactive alarm resolution, where the user inspects the most confident alarm, and the analysis updates the confidence based on the inspection. The goal of the application is to prioritize inspecting true alarms based on the confidence and minimizing the number of false alarm inspections. While the prior of the Bayesian analysis is usually crude, leading to imprecise initial alarm confidence, the quality of the alarm ranking based on the confidence gradually improves as user feedback accumulates. So a key question is, is it really optimal to ask the user to inspect the most confident alarm, as is performed in existing work [Mangal et al. 2015; Raghothaman et al. 2018; Zhang et al. 2024], given that the Bayesian model is also improving during the interaction?

Our answer is "no", and our key insight is that the problem of selecting alarms for inspection in Bayesian program analysis fits into the classical *Exploration-Exploitation* framework in decision making [Berger-Tal et al. 2014] (e.g., reinforcement learning). In this framework, *exploitation* tries to select the best option based on the current model; while *exploration* tries to improve the model by selecting an option that produces more information. Based on this framework, the existing approach of selecting the most confident alarm based on the Bayesian model can be regarded as a greedy strategy that performs exploitation only. However, interleaving exploitation with exploration is well-known to be beneficial in many scenarios [Berger-Tal et al. 2014; Mehlhorn et al. 2015; Sinha 2015]. In other words, occasionally inspecting alarms with less confidence but more information may improve the Bayesian model faster and therefore enable the user to discover true alarms in fewer interactions. So how can we select such alarms to perform exploration effectively?

The goal of performing exploration is to maximize the expected information gain in alarm selection. Therefore, we try to achieve this goal by leveraging a core property of program analysis derived from program semantics: the correlation between alarms can be estimated by the number and proximity of shared root causes, which is captured in the structure of the Bayesian model. Since labeling an alarm updates the information of alarms that are highly correlated with it, it is desirable to select an alarm that shares root causes with little inspected alarms and many uninspected alarms in the exploration stage. Based on this insight, we develop a novel algorithm to cluster analysis alarms based on the number and proximity of their shared root causes and an exploration strategy to prioritize inspecting alarms in clusters with fewer inspected alarms and more uninspected alarms.

We have developed a framework named Beer (Bayesian Exploration-Exploitation Ranker) that incorporates the above strategy and balances exploration and exploitation in interactive alarm resolution using Bayesian program analysis. Beer starts with exploitation by selecting the most confident alarm based on the Bayesian model, and performs an exploration step once it consistently recommends false alarms for inspection.

We have implemented Beer atop the Bayesian analysis framework Bingo [Raghothaman et al. 2018] and conducted experiments on three representative static analyses: datarace analysis, thread-escape analysis, and taint analysis. Compared to Bingo's default exploitation-only greedy strategy, Beer achieved an average improvement on inversion counts in terms of discovering true alarms before false alarms, Rank-100%-FP (the number of inspected false alarms when all the true alarms have been identified) and Rank-90%-FP (the number of inspected false alarms when 90% of the true alarms have been identified) of 32.10%, 21.28%, and 32.41%, respectively. These results validate the effectiveness of incorporating the Exploration-Exploitation paradigm into Bayesian program analysis. Furthermore, to isolate and verify the efficacy of our specific strategy for selecting informative exploration candidates—beyond the benefits of exploration itself—we also implemented a naive ϵ-greedy exploration strategy as a baseline. This comparison demonstrates the effectiveness of our domain-specific approach to the problem of identifying the most valuable alarms to explore.

Input relations		
$\mathsf{EV}(e, v)$	:	The statement e operate on the variable v.
$\mathsf{FH}(h)$	:	A static field may point to the heap object h at a time.
$\mathsf{HFH}(h_1, h_2)$	:	A non-static field of h_1 may point to h_2 at a time.
$\mathsf{VH}(v, h)$	:	The variable v may point to the heap object h.
Output relations		
$\mathsf{escH}(h)$	:	h may be accessed by multiple threads.
$\mathsf{escE}(e)$	:	The target of e may be accessed by multiple threads.
Derivation rules		
R_1	:	$\mathsf{escH}(h)$:- $\mathsf{FH}(h)$.
R_2	:	$\mathsf{escH}(h_2)$:- $\mathsf{escH}(h_1), \mathsf{HFH}(h_1, h_2)$.
R_3	:	$\mathsf{escE}(e)$:- $\mathsf{EV}(e, v), \mathsf{VH}(v, h), \mathsf{escH}(h)$.

Fig. 1. A simplified Datalog-based thread-escape analysis for demonstration.

In summary, our contributions are as follows:

- We introduce the Exploration-Exploitation paradigm to interactive Bayesian program analysis, overcoming the local optima problem inherent in conventional greedy strategy to efficiently find true alarms.
- We propose and implement a systematic and domain-specific exploration approach based on correlation clustering to estimate expected information gain. This method leverages program semantics to compute the correlation between alarms and groups them into clusters. This allows our strategy to quantify the information overlap between alarms, enabling it to precisely identify candidates with high expected information gain for exploration.
- We show the improvement of our approach over the conventional greedy approach on a benchmark of diverse real-world static analyses. Furthermore, through ablation studies, we demonstrate the specific efficacy of our proposed exploration heuristic.

2 Overview

In this section, we use a thread-escape analysis, a widely-used analysis in currency analysis such as datarace detection and program optimization, as an example to informally illustrate our approach. In particular, we apply a simplified, flow-insensitive thread-escape analysis in Figure 1 to a Java code example in Figure 2 to highlight the deficiencies of the conventional "exploitation-only" greedy strategy inherent in existing interactive Bayesian program analysis. We then argue for the necessity of the Exploration-Exploitation paradigm and show how our framework, Beer, effectively gains information during exploration and improves the efficiency in finding true alarms.

2.1 A Simple Thread-Escape Analysis Example

Figure 2 presents our example Java program. In this program, the `main` method of `Thread1` creates 4 objects `h0`–`h3` and stores them in an array pointed to by the static field `arr`, which makes them thread-escaping. Later, in the `run` method of `Thread1`, variables `v1`–`v3` may point to `h0`–`h3` and may create access events `e1`–`e7` on heap objects that might be visible to multiple threads.

To identify whether these accesses are thread-escaping, a thread-escape analysis is specified in Datalog (a declarative logic programming language), as shown in Figure 1. For illustration purposes, the rules are simplified from a real-world analysis. The analysis is designed to be flow-insensitive. Also, we assume that the input relations except $\mathsf{EV}(e, v)$ are computed by a pointer analysis so they can over-approximate, while $\mathsf{EV}(e, v)$ is precise as it is directly parsed from the program. The derivation rules R_1–R_3 specify the main logic of the analysis: (1) if a heap object h

```java
class T {int id; String name;}

public class Thread1 extends Thread {

  static T[] arr = new T[4]; // h10 escapes

  public static void main(String[] args) {
    if (alwaysTrue()) {
      arr[0] = new T();     // h0  escapes
      arr[1] = new T();     // h1  escapes
      arr[2] = new T();     // h2  escapes
      arr[3] = new T();     // h3  escapes
    }
    new Thread1().start();
    new Thread2().start();
  }

  static class Thread2 extends Thread {
    public void run() {
      for (int i = 0; i <= 3; i++) {
        print(arr[i].id, arr[i].name);
      }
    }
  }

  public void run() {
    if (alwaysTrue()) {
      T v1;
      if (random())
        v1 = random() ? arr[0] : arr[1];
      else {
        v1 = new T();        // v1 -> localObj
        v1.id = 0;           // e1, false
      }
      T v2 = random() ? arr[0] : arr[1];
      v2.id = 1;             // e2, true
      v2.name = "h1";        // e3, true
      T v3 = random() ? arr[2] : arr[3];
      v3.id = 2;             // e4, true
      v3 = new T();          // v3 -> localObj
      v3.id = 3;             // e5, false
      v3.name = "h3";        // e6, false
      v3.id = 4;             // e7, false
      // v3.name = "h4";     // e_i
      // more e_i if we want in Sec.2.2 & 2.3
    }
  }
}
```

Fig. 2. The motivating example program in Java.

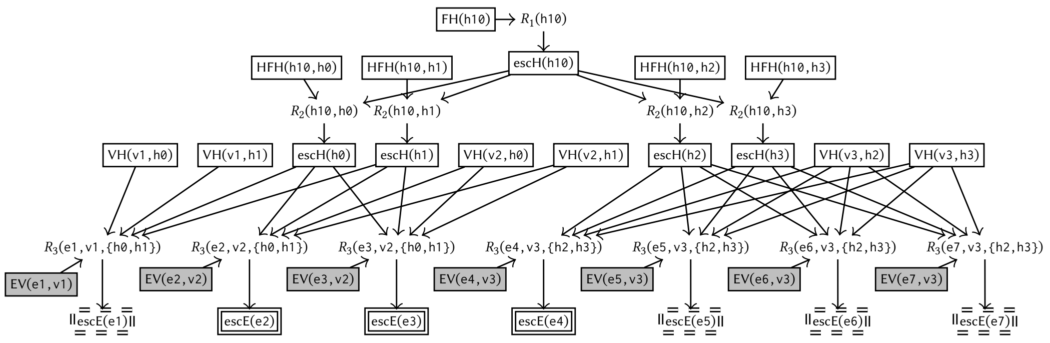


Fig. 3. The derivation graph of applying the analysis in Figure 1 on the program in Figure 2. For clarity, we represent multiple ground clauses of rules with a single expandable ground clauses. For example, R_3(e1,v1,{h0,h1}) should be expanded into R_3(e1,v1,h0) and R_3(e1,v1,h1).

may be pointed to by a static field, h may escape; (2) if a heap object h_2 may be pointed to by a field of an escaped heap object h_1, h_2 may escape; (3) if a statement e uses a variable v, and v may point to an escaped heap object h, then e may be an escape event.

Based on the rules and input facts, the Datalog inference engine repeatedly performs derivations until the set of facts converges. The engine reaches the least fixed point when a full application of the rules no longer produces any new facts, at which point the derivation process is complete.

All the derivations of the analysis can be depicted as a derivation graph in Figure 3. In this graph, we use boxed nodes to represent tuples (facts), in which the tuples in gray represent the precise input tuples in EV(e,v), and the tuples with (both solid and dashed) double borders represent alarms. Connecting these tuples are the unboxed nodes, which represent the ground clauses. For illustration purposes, we merge nodes representing ground clauses deriving the same tuple into one node. For example, R_3(e1,v1,{h0,h1}) represents two ground clauses of rule R_3: R_3(e1,v1,h0) and R_3(e1,v1,h1), and the former one fires to produce escE(e1) because the statement e1 operates on variable $v1$ as specified by tuple EV(e, v), which points to h0 as specified by VH(v1,h0), and h0 is an escaped object as specified by escH(h0).

To guarantee soundness (not missing any potential risks), this set of Datalog rules is designed to over-approximate. The trade-off of this design is the inevitable introduction of some false positives that do not correspond to the program's actual execution behavior. The border style of an alarm tuple indicates whether it is true: true positives (like escE(e2)) are shown with a solid double border while false positives (like escE(e1)) have a dashed double border. Besides imprecise inputs, a primary source of this imprecision is the analysis's flow-insensitive nature, wherein it treats the program as an unordered set of statements rather than a sequential flow of execution. For example, escE(e1) is a false positive which is produced by confusing the control-flow of statements involving v1 and thus falsely assuming v1 may point to escaped objects h0 or h1. On the other hand, escE(e2) is a true positive as at statement e2, variable v2 points to h0 or h1 which are both escaped objects.

In this example, the worst-case scenario requires the user to inspect all 7 alarms to identify the three true alarms. With a random inspection strategy, the expected number of inspected alarms is 6. In practice, the ratio of wasted user effort grows as the false positive rate grows. We next describe how Bayesian program analysis addresses this problem through interactive alarm resolution.

2.2 Original Interactive Bayesian Program Analysis and Its Local Optima Problem

To accelerate the process of locating true positive alarms, researchers have proposed interactive Bayesian program analysis [Mangal et al. 2015; Raghothaman et al. 2018; Zhang et al. 2017b]. This approach converts the derivation graph of a Datalog analysis into a probabilistic graphical model known as a Bayesian network. This transformation enables the computation of a quantitative confidence score (i.e., a probability) for each tuple (including reported alarms) and systematic integration of user feedback. The methodology operates through an interactive loop: The system presents the alarm with the highest current confidence score to the user for inspection; the user then provides feedback (`true` or `false`), which the system incorporates as posterior information to update the probabilities of other alarms throughout the network. The latter facilitates a re-ranking of the alarms, with the objective of prioritizing the presentation of true alarms to the user.

In our example, the Datalog derivation graph is converted into a Bayesian network, shown in Figure 4. In the conversion process, each tuple and ground clause becomes a Bernoulli random variable, representing whether the tuple or the ground clause holds. To quantify the inherent uncertainty constraints between the tuples and ground clauses, which are brought by imprecision from over-approximation, we assign a prior conditional probability p_{R_i} to each derivation rule (R_1, R_2, and R_3), signifying our belief that "given that all the preconditions are true, a rule is triggered at the probability of p_{R_i}." The probability p_{R_i} can be learned from other labeled alarms [Kim et al. 2022; Raghothaman et al. 2018]. For demonstration purposes, we set all the prior probabilities to 0.95. We use escE(e1) and its relevant ancestors including R_3(e1,v1,h0) and R_3(e1,v1,h1) as an example. The conditional probability of R_3(e1,v1,h0) is:

$$\Pr(R_3(\mathtt{e1},\mathtt{v1},\mathtt{h0}) \mid \mathsf{EV}(\mathtt{e1},\mathtt{v1}) \land \mathsf{VH}(\mathtt{v1},\mathtt{h0}) \land \mathsf{escH}(\mathtt{h0})) = 0.95 \quad (1)$$

$$\Pr(R_3(\mathtt{e1},\mathtt{v1},\mathtt{h0}) \mid \neg(\mathsf{EV}(\mathtt{e1},\mathtt{v1}) \land \mathsf{VH}(\mathtt{v1},\mathtt{h0}) \land \mathsf{escH}(\mathtt{h0})\ ) = 0 \quad (2)$$

Since the input tuples in Figure 1 (FH, HFH and VH) are computed by an over-approximated pre-analysis, we also assign prior probabilities of 0.95 to them. On the other hand, tuples in relation

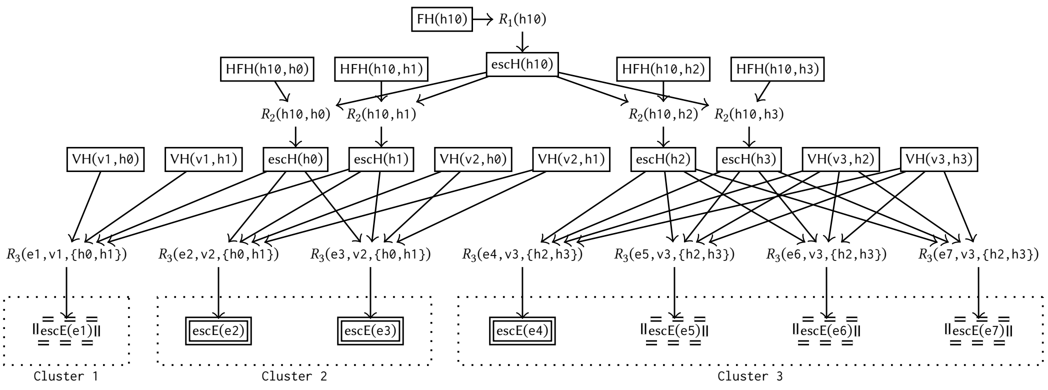


Fig. 4. The underlying graph of the Bayesian network converted from the derivation graph from Figure 3. The EV tuples, which are precise, are deprecated in the Bayesian network. Similarly for clarity, we represent multiple ground clauses of rules with a single expandable ground clause. For example, R_3(e1,v1,{h0,h1}) should be expanded into R_3(e1,v1,h0) and R_3(e1,v1,h1).

EV always hold so that they can be omitted from the conditional probability equation and the Bayesian network:

$$\Pr(R_3(\mathtt{e1},\mathtt{v1},\mathtt{h0}) \mid \mathrm{VH}(\mathtt{v1},\mathtt{h0}) \wedge \mathrm{escH}(\mathtt{h0})) = 0.95 \tag{3}$$

$$\Pr(R_3(\mathtt{e1},\mathtt{v1},\mathtt{h0}) \mid \neg(\mathrm{VH}(\mathtt{v1},\mathtt{h0}) \wedge \mathrm{escH}(\mathtt{h0})\,) = 0 \tag{4}$$

Similarly, we have the conditional probability of R_3(e1,v1,h1). Then, if either R_3(e1,v1,h0) or R_3(e1,v1,h1) is true, their child escE(e1) is true with probability 1:

$$\Pr(\mathrm{escE}(\mathtt{e1}) \mid R_3(\mathtt{e1},\mathtt{v1},\mathtt{h0}) \vee R_3(\mathtt{e1},\mathtt{v1},\mathtt{h1})) = 1 \tag{5}$$

$$\Pr(\mathrm{escE}(\mathtt{e1}) \mid \neg(R_3(\mathtt{e1},\mathtt{v1},\mathtt{h0}) \vee R_3(\mathtt{e1},\mathtt{v1},\mathtt{h1}))\,) = 0 \tag{6}$$

Similar patterns are applied to each node in the derivation graph.

After the conversion from the derivation graph to the Bayesian network, the Bayesian program analysis begins its interactive phase with the user. We take a possible trace of the interaction in Table 1a as an example. The analysis first ranks all alarms by their marginal probabilities inferred [Murphy et al. 1999] on the Bayesian network, as specified in column Step 0, and presents one of the alarms with the highest probability for inspection. Here all the alarms have the same probability, and the analysis yields escE(e4). Upon receiving the user's feedback that it is a true positive, the analysis performs a new round of marginal inference, propagating the feedback as posterior information to update the posterior probabilities of all other alarms, as specified in column Step 1. However, because the priors in Bayesian analysis are often coarse, the resulting the alarm probabilities during early steps can be imprecise, causing the highest-ranked alarm to be a false positive (escE(e5)) in this case. But in Step 2, the marginal inference will lower the probabilities of the four ancestors of escE(e5), namely escH(h2), escH(h3), VH(v3,h2), and VH(v3,h3), which in turn lowers the probabilities of the other two false alarms escE(e6) and escE(e7) since they share the same ancestors. Note that these three false alarms are derived for the same reason that is captured by the common ancestors: The analysis falsely thinks that the statements can access escaping objects h2 and h3 via v3. Decreasing the probabilities of the false alarms lifts the rank of true alarms escE(e2) and escE(e3). Then, by getting the feedback of "escE(e2) is true", in Step 3 the analysis further lifts the rank of escE(e3) since it shares common ancestors with escE(e2) and

Table 1. Two possible interaction traces for the motivating example. In red shadow are true alarms. The alarm selected for the user feedback is shown in bold.

(a) An ideal trace of the original Bayesian analysis: lucky initialization that prioritizes a true alarm among those with the highest probability enables finding all true alarms within only four interactions.

Step Feedback	0 /		1 escE(e4)		2 ¬escE(e5)		3 escE(e2)		4 escE(e3)		- -	
Rank	Prob.	Alarm	Prob.	Alarm	Prob.	Alarm	Prob.	Alarm	Prob.	Alarm	Prob.	Alarm
1	0.8714	**escE(e4)**	0.9861	**escE(e5)**	0.9656	**escE(e2)**	0.9861	**escE(e3)**	0.9765	escE(e1)	-	-
2	0.8714	escE(e5)	0.9861	escE(e6)	0.9656	escE(e1)	0.9760	escE(e1)	0.9565	escE(e6)	-	-
3	0.8714	escE(e6)	0.9861	escE(e7)	0.9656	escE(e3)	0.9565	escE(e6)	0.9565	escE(e7)	-	-
4	0.8714	escE(e7)	0.9656	escE(e2)	0.9565	escE(e6)	0.9565	escE(e7)	-	-	-	-
5	0.8714	escE(e2)	0.9656	escE(e1)	0.9565	escE(e7)	-	-	-	-	-	-
6	0.8714	escE(e1)	0.9656	escE(e3)	-	-	-	-	-	-	-	-
7	0.8714	escE(e3)	-	-	-	-	-	-	-	-	-	-

(b) When the original Bayesian program analysis falls into the local optima unluckily, it consistently recommends false alarms for inspection (step 3–4). here, exploration (step 5) is needed to maximize the expected information gain by inspecting a alarm from the least known cluster (escE(e3) from Cluster 2) to get the new information about their nearby tuples, and finally optimizes the efficiency in finding all the true alarms.

Step Feedback	0 /		1 ¬escE(e1)		2 escE(e4)		3 ¬escE(e5)		4 ¬escE(e6)		**5** **escE(e3)**	
Rank	Prob.	Alarm	Prob.	Alarm	Prob.	Alarm	Prob.	Alarm	Prob.	Alarm	Prob.	Alarm
1	0.8714	**escE(e1)**	0.2332	**escE(e4)**	0.9861	**escE(e5)**	0.9565	**escE(e6)**	0.9504	escE(e7)	0.9643	**escE(e2)**
2	0.8714	escE(e2)	0.2332	escE(e5)	0.9861	escE(e6)	0.9565	escE(e7)	0.6730	escE(e2)	0.9504	escE(e7)
3	0.8714	escE(e3)	0.2332	escE(e6)	0.9861	escE(e7)	0.6730	escE(e2)	0.6730	**escE(e3)**	-	-
4	0.8714	escE(e4)	0.2332	escE(e7)	0.6730	escE(e2)	0.6730	escE(e3)	-	-	-	-
5	0.8714	escE(e5)	0.1626	escE(e2)	0.6730	escE(e3)	-	-	-	-	-	-
6	0.8714	escE(e6)	0.1626	escE(e3)	-	-	-	-	-	-	-	-
7	0.8714	escE(e7)	-	-	-	-	-	-	-	-	-	-

the ancestors have higher probabilities than before. In the end, the user only needs to check 4 alarms to find all the true alarms, showing the effectiveness of Bayesian program analysis.

However, this purely exploitative greedy strategy is fragile and can easily fall into a "local optima problem". In the "unlucky" trace shown in Table 1b, the process begins with the analysis selecting escE(e1), whose negative feedback suppresses the ranks of true positives escE(e2) and escE(e3). Subsequently, a single piece of positive feedback for escE(e4) causes the probabilities of its highly correlated false-positive siblings (escE(e5), escE(e6), escE(e7)) to surge, placing their probability higher than the other true alarms. The local optimum is now set: even as the user repeatedly flags escE(e5) and escE(e6) as false, the greedy strategy continues to pick the next top-ranked alarm with the shared ancestor causes, escE(e7). Even worse, if there are more alarms sharing the same ancestors, the analysis will always prioritize these alarms until they are all marked false. In this situation, the analysis becomes "stuck", repeatedly acquiring redundant information about the shared ancestors of escE(e4),escE(e5),escE(e6),escE(e7), ... instead of "exploring" other regions for new information gain. This exposes the fundamental flaw of pure exploitation—it lacks a mechanism to escape an incorrectly high-confidence region.

To break this impasse, the analysis must perform **exploration** once it consistently recommends false alarms for inspection. For example, for `Step` 5 in Table 1b when the analysis consistently recommends $K_{trigger}$ false alarms (here we set $K_{trigger} = 2$), it is beneficial for the analysis to select escE(e3), though it has a relatively low current probability and rank. The feedback on it brings new and distinct information that at least one of the conjunctions $\mathtt{VH(v2,h0)} \land$ escH(h0) and $\mathtt{VH(v2,h1)} \land$ escH(h1) should be true, and increases the probability of the true alarm escE(e2)

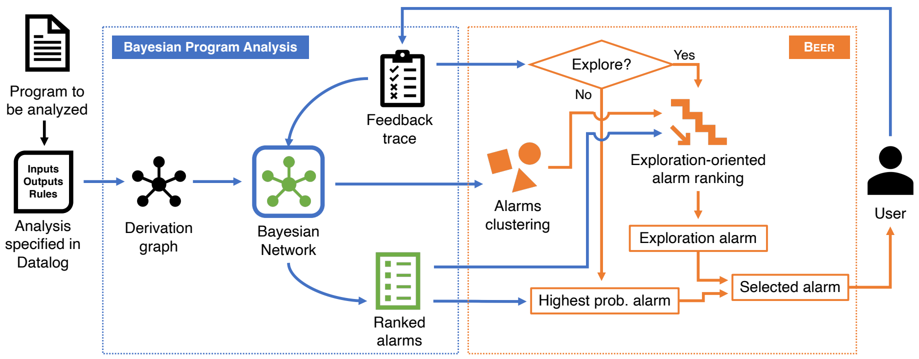


Fig. 5. Overview of BEER which introduces Exploration-Exploitation into Bayesian program analysis.

which has shared ancestors. This case highlights the necessity of strategic exploration to obtain new information gain, further improve the model, and accelerate the discovery of true positives.

Given that exploration is necessary, the next critical question is: Which alarm should be selected during an exploration step to maximize the gain of new information?

A natural first thought is to apply classic strategies from the machine learning community such as ϵ-greedy [Sutton et al. 1998]. However, these strategies ignore correlations between alarms incurred by the analysis semantics, making them less effective. We next explain our exploration strategy and the overall framework atop it.

2.3 Our Approach

Our core insight to perform exploration is that alarms correlate with each other through shared ancestors in analysis derivation, and inspecting alarms in large alarm clusters with high uncertainties yields the most information. Intuitively, the more ancestors two alarms share and the closer these ancestors are to the alarms in the derivation, the more likely it is that inspecting one alarm yields more information about the other. This observation reflects semantic dependencies and is captured in the Bayesian network. So we can cluster alarms based on the number and proximity of shared ancestors in the derivation and then estimate the expected information gain of inspecting an alarm in a cluster based on the size of the cluster and how each alarm is labeled in the cluster.

Based on the above insight, we propose the BEER framework, whose overall workflow is depicted in Figure 5. The core of BEER is to augment the traditional exploitation-only strategy with a domain-specific exploration mechanism. The method consists of three main steps:

Alarm Clustering. Before the interaction with the user begins, BEER pre-analyzes the structure of the Bayesian network and clusters alarms based on the number and proximity of their shared ancestors. Concretely, for every two alarms, BEER counts the number of shared ancestors within a given distance, which is specified by an expert or learned on a training set. Then, by considering the number of shared ancestors from large to small, BEER groups alarms into clusters.

For example, in Figure 4, we set the distance threshold to one, which blocks the correlation incurred by escH(h10), HFH tuples, and FH(h10), as they incur weak correlation on many alarms. Note that we do not consider nodes representing ground clauses in common ancestors or distance calculation, as their impact is captured in the analysis facts they derive. BEER groups escE(e2) and escE(e3) into Cluster 2 and escE(e4), escE(e5), escE(e6) and escE(e7) into Cluster 3 as shown in Figure 4, because they share four ancestors at a distance of 1, which is the largest in terms of the number of shared ancestors for any two alarms. Although escE(e1) shares two ancestors with

$$
\begin{array}{rlrlrl}
\text{(program)} & C \coloneqq c^\star & \text{(constraint)} & c \coloneqq l \coloneq l^\star & \text{(literal)} & l \coloneqq r(a^\star) \\
\text{(relation)} & r \in \mathbf{R} & \text{(argument)} & a \coloneqq v \mid d & \text{(variable)} & v \in \mathbf{V} = \{x, y, \cdots\} \\
\text{(constant)} & d \in \mathbf{D} = \{0, 1, \cdots\} & \text{(tuple)} & t \in \mathbf{T} = \mathbf{R} \times \mathbf{D}^\star & &
\end{array}
$$

Fig. 6. Syntax of Datalog.

escE(e2) and escE(e3), it is assigned to a separate cluster because its relation to them is weaker than the tight bond between escE(e2) and escE(e3).

Choosing Whether to Explore. During the interaction, Beer monitors the user's feedback trace. When it observes that the last $K_{trigger}$ consecutive alarms in the trace were all marked "false", it infers that the greedy strategy is likely stuck in a local optimum and triggers an exploration step.

While $K_{trigger}$ can be specified by an expert or learnt on a training set, we set $K_{trigger} = 2$ in our example for illustration. In the trace from Table 1b, this condition is met before `Step 5`, as the feedback for escE(e5) (`Step 3`) and escE(e6) (`Step 4`) are both false. The system therefore triggers an exploration step.

Exploration-Oriented Alarm Ranking. During an exploration step, our heuristic prioritizes clusters that are both uncertain and informative. Specifically, Beer orders clusters by *(number of confirmed alarms, - number of unconfirmed alarms)* from low to high and selects the top cluster. Here, we prioritize considering the number of confirmed alarms as even confirming one alarm would give much information to the cluster. Such a number reflects the amount of uncertainty in the cluster. In addition, Beer prioritizes clusters with no confirmed true alarms as these clusters are poorly understood and are unlikely to be selected in the exploitation phase. Finally, the number of unconfirmed alarms reflects how many alarms would gain information by confirming an alarm in the cluster. Once the cluster is chosen, Beer would select an alarm from it based on a certain strategy. In our evaluation, Beer chooses the least confident alarm according to the Bayesian analysis as it is the most unlikely one to be chosen in the exploitation phase.

In `Step 5` of Table 1b, `Cluster 2` is the only completely unexplored cluster. Beer can select either escE(e2) or escE(e3) from it as they have the same probability. As the trace shows, the system explores escE(e3). This move is highly effective: it not only uncovers a true positive directly but also provides the key information that at least one of the conjunctions—either VH(v2,h0) $\wedge$ escH(h0) or VH(v2,h1) $\wedge$ escH(h1)—is valid. This, in turn, boosts the probability and rank of the other related true alarm, escE(e2), demonstrating a successful exploration that breaks the local optimum. Note that selecting escE(e2) would yield similar results. Once exploration is finished, Beer switches back to exploitation in the original Bayesian analysis framework.

As shown in the above example and discussed in the end of Section 2.2, the key to successful exploration is to predict the alarm with high expected information gain and use it to guide exploration. We next formulate the problem and propose our practical approximate solution in Section 4, and then detail its implementation in Section 5.

3 Preliminaries

Before getting into the details of our approach, we introduce some preliminaries which contain useful definitions and symbols to help demonstrate our approach. We first introduce the syntax and semantics of Datalog in Section 3.1. Then, we introduce how to write program analysis in Datalog in Section 3.2. Finally, we introduce Bayesian program analysis in Section 3.3.

$$\llbracket C \rrbracket \in \mathcal{P}(\mathbf{T}) = \text{lfp } F_C \qquad F_C(T) \in \mathcal{P}(\mathbf{T}) = T \cup \bigcup \{ f_c(T) \mid \forall c \in C \}$$
$$f_{l_0 \text{:-} l_1,\dots,l_n}(T) \in \mathcal{P}(\mathbf{T}) = \{ \bar{\sigma}(l_0) \mid \forall \sigma \in \mathbf{V} \mapsto \mathbf{D}.\ \forall k.\ 1 \le k \le n.\ \bar{\sigma}(l_k) \in T \}$$
$$\text{where} \quad \bar{\sigma}\big(r(a_1, a_2, \dots, a_m)\big) = r\big(\sigma(a_1), \sigma(a_2), \dots, \sigma(a_m)\big)$$

Fig. 7. Denotational Semantics of Datalog.

3.1 Datalog: Syntax and Semantics

Datalog is a declarative logic programming language that makes it suitable to declare logical rules and therefore develop program analyses. Because of its declarative nature, developers only need to focus on describing analysis specifications as (derivation) rules, abstracting away the complexities of lower-level operational details. Many program analysis frameworks, like Doop [Bravenboer and Smaragdakis 2009] and Chord [Naik 2011], use Datalog to express the analysis logic.

Figure 6 shows the syntax of Datalog.[1] The topmost component is the Datalog *program*, which is made up of constraints. Every *constraint* (or *rule*) c consists of a head, which is a single literal, and a body, which is a conjunction of one or more literals. For example, $\mathsf{escH}(h)$:- $\mathsf{FH}(h)$ is a constraint with $\mathsf{escH}(h)$ as its head and $\mathsf{FH}(h)$ as its body. A *literal* is made up of a relation name and a list of arguments. For example, both $\mathsf{escH}(h)$ and $\mathsf{FH}(h)$ are literals. An *argument* can either be a variable or a constant. Literals made up of only constant arguments are called *tuples*. For example, $\mathsf{escE}(\mathtt{e1})$ is a tuple while $\mathsf{escE}(h)$ is not. [2]

Figure 7 shows the denotational semantics of Datalog. Every Datalog program finally produces a set of tuples, which is computed by applying all constraints (F_C) until the fixed point is reached. F_C is defined as applying all single rule $c ::= l_0$:- $l_1, \dots, l_n \in C$ once. f_c is the application of a single rule. The rule will be instantiated if there exists a substitution σ such that all tuples in its body ($\bar{\sigma}(l_k)$) hold. Then the instantiated head will be the output of f_c. For example, for substitution $\sigma_0 = \{h \mapsto \mathtt{e1}\}$, since $\bar{\sigma}_0(\mathsf{FH}(h)) = \mathsf{FH}(\mathtt{e1})$ holds, $\bar{\sigma}_0(\mathsf{escH}(\mathtt{e1}))$ will be appended to the result set.

3.2 Datalog Program Analysis

A program analysis implemented in Datalog contains two parts:

- A set of constraints specifying the analysis specification. For a specific analysis, this part is unchanged when applying to different programs;
- A set of input tuples representing the intermediate-representation (IR) of the analyzed program (For example, the assignments in the program). This part is different for each program.

Given these two parts, the Datalog engine will apply rules according to the semantics mentioned in Section 3.1 to generate new tuples until it reaches the fixed point. The information the user focuses on (for example, whether a datarace exists) will be denoted by certain output relations.

The calculation process (derivation) of the Datalog engine can be formed as a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. where $\mathcal{V}$ stands for tuples and ground clauses in the derivation; while $\mathcal{E}$ stands for derivations between nodes: For every instantiation of rule c based on substitution σ, it creates a special node $gc_{c,\sigma}$ as the ground clause, and creates edges from the body of c to the ground clause, expressed as $\{(\bar{\sigma}(l_k), gc_{c,\sigma}) \mid \forall k \in \{1, \dots, n\}\}$ and an edge from the ground clause to the head, expressed as $(gc_{c,\sigma}, \bar{\sigma}(l_0))$. Figure 3 shows the derivation graph derived from the Datalog program in Figure 1.

[1]"$\star$" denotes the Kleene star, which stands for zero or more occurrences of a certain term.

[2]For readability, we use letters like e1 instead of integers to represent constants.

3.3 Bayesian Program Analysis

We adopt the approach from Bingo [Raghothaman et al. 2018] to map the derivation graph $\mathcal{G}$ to a Bayesian network. Since Bayesian networks require a directed acyclic graph, Bingo prunes ground clauses to break cycles. This yields an acyclic underlying graph $G = (V, E)$ with $V \subseteq \mathcal{V}$ and $E \subseteq \mathcal{E}$. Figure 4 shows the underlying graph of the Bayesian network converted from the derivation graph in Figure 3. After the conversion, every node $v \in V$ represents a Bernoulli random variable, and every edge $e \in E$ represents a conditional probabilistic dependency. For each ground clause $gc_{c,\sigma} : \bar{\sigma}(l_0) \text{ :- } \bar{\sigma}(l_1), ..., \bar{\sigma}(l_n)$, we define that $\Pr(gc_{c,\sigma} \mid \bigwedge_{i=1,...,n}\bar{\sigma}(l_i)) = p_c$ (p_c is the probability assigned to rule c) and $\Pr(gc_{c,\sigma} \mid \neg(\bigwedge_{i=1,...,n}\bar{\sigma}(l_i))) = 0$. For each tuple, $\Pr(\bar{\sigma}(l_0) \mid \bigvee_{(gc_{c',\sigma'},\bar{\sigma}(l_0))\in E}\, gc_{c',\sigma'}) = 1$ and $\Pr(\bar{\sigma}(l_0) \mid \neg(\bigvee_{(gc_{c',\sigma'},\bar{\sigma}(l_0))\in E}\, gc_{c',\sigma'})) = 0$. Notice that $\bar{\sigma}(l_0)$ may be derived from other substitutions σ' or constraints c', e.g., in Equation 1, 3, and 5.

With the Bayesian network, we can infer the posterior marginal probability for each alarm. For the clarity in the following sections, we establish some common notations: A denotes the alarm set, which is partitioned into three disjoint subsets: U, the subset of the alarms that have not been labeled by the user; T, the set of the alarms labeled `true` by the user; F, the set of the alarms labeled `false`. We also define a function Pr to calculate the posterior marginal probability of alarm a given the user feedback (observation) trace *obsTrace* which serves as a partial assignment of the alarms:

$$\begin{gathered}\textsc{Pr}(a, \mathit{obsTrace}) = \Pr(a \mid \mathit{alarm}_1 = \mathit{feedback}_1 \wedge \mathit{alarm}_2 = \mathit{feedback}_2 \wedge \ ...)\\ \text{where } a \in U,\ \mathit{alarm}_i \in A \setminus U,\ \mathit{feedback}_i \in \{\texttt{true}, \texttt{false}\},\\ \mathit{obsTrace} = [(\mathit{alarm}_1, \mathit{feedback}_1), (\mathit{alarm}_2, \mathit{feedback}_2), ...]\end{gathered}$$

We take Table 1a as an example. Before `Step 3`, $\mathit{obsTrace} = [(\text{escE(e4)}, \texttt{true}), (\text{escE(e5)}, \texttt{false})]$ and $\textsc{Pr}(\text{escE(e3)}) = \Pr(\text{escE(e3)} \mid \text{escE(e4)} \wedge \neg\text{escE(e5)}) = 0.9656$. After `Step 3`, $\mathit{obsTrace} = [(\text{escE(e4)}, \texttt{true}), (\text{escE(e5)}, \texttt{false}), (\text{escE(e2)}, \texttt{true})]$ and $\textsc{Pr}(\text{escE(e3)}) = \Pr(\text{escE(e3)} \mid \text{escE(e4)} \wedge \neg\text{escE(e5)} \wedge \text{escE(e2)}) = 0.9861$.

4 Estimating Expected Information Gain in Exploration

As outlined in Section 2.3, the key to successful exploration is to use Expected Information Gain (EIG) as guidance. This section formally defines EIG and the problem of finding the alarm suitable for exploration (which is the one that maximizes the EIG), and then introduces an approximate solution for it, which is integrated in Section 5. Detailed derivations are provided in Appendix A.

Given a feedback trace *obsTrace*[3], for an exploration candidate alarm X, its expected information gain is quantified by the Kullback-Leibler divergence (D_{KL}) between the distributions of all the random variables V except X (denoted as $V\setminus\{X\}$, where $\bar{S} = S\setminus\{X\}$ for any set S) before and after querying X, and our goal is to find the variable X' with the maximum EIG:

Definition 4.1 (The Exploration Alarm Selection Problem). *Our goal in the exploration phase is to find an alarm* $X \in \mathit{argmax}_{X' \in U} \mathit{EIG}_{\bar{V}}(X')$ *where* $\mathit{EIG}_{\bar{V}}(X) = E_X[\mathit{IG}(\bar{V}, X)]$ *and* $\mathit{IG}(\bar{V}, x) = D_{KL}(\mathit{Pr}_{\bar{V}|X=x} \parallel \mathit{Pr}_{\bar{V}})$.

However, calculating the KL divergence for $|\bar{V}|$ Bernoulli variables typically requires traversing $2^{|\bar{V}|}$ states, rendering it computationally intractable. We therefore employ approximations based on the characteristics of Bayesian networks in Bayesian program analysis.

[3]We omit the notation "conditional on *obsTrace*" henceforth, as it is implicitly assumed in all expressions if not explicitly specified. For example, $\text{EIG}_{\bar{V}}(X)$ denotes the EIG of $\bar{V}$ conditional on *obsTrace*, and $\Pr_{\bar{V}|X=x}$ represents the probability of $\bar{V}$ conditional on $X = x$ and *obsTrace*.

4.1 Clustering for Local EIG Estimation with Lower Complexity

Efficient EIG estimation necessitates decomposing the global information gain into manageable local components. Our decomposition strategy is informed by the observation that mutual information in Bayesian networks diminishes with increasing node distance and decreasing number of shared ancestors (demonstrated in Appendix A.1). Accordingly, we cluster alarms with their shared ancestors based on these metrics into clusters $\{C_1, \dots, C_n\}$, leaving unclustered non-alarm nodes in C_0, such that $V = \bigcup_{i=0}^{n} C_n$ and $C_i \cap C_j = \emptyset$ for all i, j such that $i \neq j$. Given the rapid decay of mutual information with longer distance and fewer shared ancestors, we expect inter-cluster mutual information to be minimal compared to intra-cluster entropy. This leads to the following formal assumption:

Assumption 4.2. *The mutual information $I(M; N) = H(M) - H(M \mid N)$ between a cluster and the rest of the variables is negligible. Formally, $I(C_i; V \backslash C_i) \approx 0$.*

Based on the chain rule of entropy ($\mathrm{H}(\bar{C}_0, \dots, \bar{C}_n) = \mathrm{H}(\bar{C}_0) + \sum_{i=1}^{n} \mathrm{H}(\bar{C}_i \mid \bar{C}_0, \dots, \bar{C}_{i-1})$) and Assumption 4.2, the global entropy can be approximated as the sum of cluster local entropies:

Lemma 4.3. *$H(\bar{V}) = H(\bar{C}_0, \dots, \bar{C}_n) \approx \sum_{i=0}^{n} H(\bar{C}_i)$.*

From Assumption 4.2 and Lemma 4.3, we can derive:

Property 4.4. *$EIG_{\bar{V}}(X) \approx E_X[H(\bar{C}_X) - H(\bar{C}_X \mid X)] = EIG_{\bar{C}_X}(X)$, where $X \in C_X$.*

This implies that, under our assumption, the global EIG of X can be approximated by its local EIG within its assigned cluster C_X, representing the expected reduction in entropy for the remaining variables $\bar{C}_X$ in that cluster. This leads to the first part of our solution: *we cluster alarms with their shared ancestors based on their distances to the shared ancestors and the number of shared ancestors to divide the global information gain into manageable local components.*

4.2 Estimating Local EIG using Feedback Trace

To estimate the local EIG of $\bar{C}_X$, we calculate the entropy of $\bar{C}_X$ before and after labeling X and utilize statistics derived from the feedback trace: the certainty of $\bar{C}_X$ is captured by the number of labeled true alarms $t := |T_{\bar{C}_X}|$ and labeled false alarms $f := |F_{\bar{C}_X}|$ in C_X, while the uncertainty is captured by the number of unlabeled alarms $u := |U_{\bar{C}_X}|$.

To ensure the computational feasibility of the approximation, we introduce an assumption, which is grounded in our empirical observations. We observe that approximately 90% of the tuples in Bayesian networks have a unique predecessor ground clause. Because the majority of tuples are derived from a single rule, we formalize the assumption as follows:

Assumption 4.5. *The cardinality of the predecessor clause set for any Bernoulli variable b is at most 1, formally, $|\{gc \mid \forall gc.(gc, b) \in E\}| \leq 1$.*

We define nodes in a cluster whose ancestors are not in the cluster as the root ancestors of the cluster. Empirical observations of derivation graphs reveal that intermediate nodes, other than the root ancestors and leaf nodes (alarms), constitute less than 20% of the cluster. Consequently, given their rarity and the fact that their uncertainty is largely inherited from root ancestors, we ignore intermediate nodes and posit the following structural simplification:

Assumption 4.6. *Each cluster C_i is modeled as a three-layer structure (see Figure 8) comprising* r *shared root ancestors $R_1, \dots, R_r$,* a *corresponding alarms $alarm_1, \dots, alarm_a$ and* a *ground clauses gc_j : $alarm_j$:-$R_1, \dots, R_r$. $\forall j. 1 \leq j \leq a. Pr(gc_j \mid \bigwedge_{i=1,\dots,r} R_i) = p_{rule}$, $Pr(alarm_j \mid gc_j) = 1$.*

Based on the number of labeled alarms and unlabeled alarms t, f, u, the entropy within $\bar{C}_X$ can be expressed as:

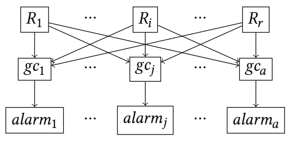


Fig. 8. The simplified model in Assumption 4.6.

Property 4.7. *$H(\bar{C}_x \mid t, f, u) = \sum_{i=1}^{r} H(R_i \mid t, f, u) + u \cdot H(p_{rule}) \cdot \prod_{i=1}^{r} Pr(R_i \mid t, f, u)$.*

This indicates that cluster entropy decomposes into the entropy of root ancestors and the conditional entropy of unlabeled alarms.

If $t := |T_{\bar{C}_X}| \geq 1$, then $\Pr(R_i \mid t \geq 1, f, u) = 1$ and $\mathrm{H}(R_i \mid t \geq 1, f, u) = 0$. Consequently:

Property 4.8. *If $t \geq 1$, then $H(\bar{C}_X \mid t \geq 1, f, u) = u \cdot H(p_{rule})$, and $EIG_{\bar{V}}(X) \approx EIG_{\bar{C}_X}(X) = E_X[H(\bar{C}_X) - H(\bar{C}_X \mid X)] = 0$.*

This implies that if a cluster contains a labeled true alarm, its entropy depends solely on the number of unlabeled alarms. Consequently, inspecting an alarm in it yields no additional information gain other than the information on it. This observation establishes the second part of our solution: *for two alarms X_1 and X_2, if the cluster C_{X_1} containing X_1 contains a labeled true alarm, then the EIG of X_1 is approximately no more than that of X_2.*

Having addressed the case with labeled true alarms, we now consider cases where no labeled true alarm exists in $\bar{C}_X$ ($t = 0$). Without losing generality, we introduce the following assumption to simplify the evaluation of the impact of false observations and unlabeled warnings on entropy:

Assumption 4.9. *The priors of R_is are equal: $Pr(R_1) = ... = Pr(R_r)$.*

This leads to the following property:

Property 4.10. *If $t = 0$, then $H(\bar{C}_X \mid t = 0, f, u) \sim f \cdot (1 - p_{rule})^{\frac{f}{r}} + (1 - p_{rule})^{f} \cdot u$, which decays exponentially as f increases and slightly grows linearly as u increases when f is fixed.*

Exploration is typically triggered after encountering multiple false alarms, where the posterior probabilities of other unlabeled alarms are low. Also, false alarms are prevalent due to static analysis's over-approximation. Thus, we introduce the following assumption to simplify our estimation:

Assumption 4.11. *$P(X = True \mid t, f, u) \approx 0$.*

Then we have:

Property 4.12. *$EIG_{\bar{V}}(X) \sim H(\bar{C}_X \mid t = 0, f, u) - H(\bar{C}_X \mid t = 0, f + 1, u)$, which also decays exponentially as f increases and slightly grows linearly as u increases when f is fixed.*

Property 4.12 shows that the factor f dominates u, which determines the third part of our solution: *for two alarms X_1 and X_2, if the number of labeled false alarms f_{X_1} in C_{X_1} is no less than f_{X_2} in C_{X_2}, then the EIG of X_1 is approximately no more than that of X_2. If $f_{X_1} = f_{X_2}$, then if the number of unlabeled alarms u_{X_1} in C_{X_1} is no more than u_{X_2} in C_{X_2}, then the EIG of X_1 is approximately no more than that of X_2.*

In summary, our solution is: we cluster alarms based on the distance to and the number of their shared ancestors, and select the least alarm X in lexicographical order using a 3-tuple sort key $([t_X > 0], -f_X, u_X)$[4], where earlier components dominate the later ones, during exploration.

[4]Here, [] denotes Iverson Bracket that $[e] = \begin{cases} 1 & \text{if } e = \mathit{true} \\ 0 & \text{if } e = \mathit{false} \end{cases}$.

Algorithm 1 Overall interaction.

```
Require: The alarm set $A$, the Bayesian network $G$.
1: procedure INTERACTION($A, G$)
2:     $C \leftarrow$ ALARMCLUSTERING($A, G$)                                ▷ Section 5.2
3:     $K_{counter} \leftarrow 0, U \leftarrow A$, $obsTrace \leftarrow$ an empty list
4:     while the termination condition (e.g., $|U| < |A| \cdot 80\%$) is not satisfied do
5:         if $obsTrace$ is empty then
6:             $selectedAlarm \leftarrow \text{argmax}_{a \in U}$ PR($a, obsTrace$)
7:         else if $obsTrace.last.feedback$ = true then
8:             $K_{counter} \leftarrow 0$
9:             $selectedAlarm \leftarrow \text{argmax}_{a \in U}$ PR($a, obsTrace$)
10:        else
11:            $K_{counter} \leftarrow K_{counter} + 1$
12:            if $K_{counter} < K_{trigger}$ then
13:                $selectedAlarm \leftarrow \text{argmax}_{a \in U}$ PR($a, obsTrace$)
14:            else
15:                $K_{counter} \leftarrow 0$
16:                $selectedAlarm \leftarrow$ SELECTFOREXPLORATION($C, U, obsTrace$)    ▷ Section 5.3
17:        $feedback \leftarrow$ REQUESTFEEDBACK($selectedAlarm$)
18:        $U \leftarrow U \setminus \{selectedAlarm\}$
19:        $obsTrace$.append(($selectedAlarm, feedback$))
```

5 The BEER Framework

Section 5.1 provides an implementation of the interactive alarm resolution process of a BEER-enhanced Bayesian program analysis. The exploration module, which is the core of this work, consists of two key components: Alarm Clustering in Section 5.2 implementing Section 4.1 and Exploration-Oriented Alarm Ranking in Section 5.3 implementing Section 4.2.

5.1 Workflow of BEER-Enhanced Bayesian Program Analysis

Algorithm 1 describes the complete workflow of the BEER-enhanced Bayesian program analysis. The process begins by clustering alarms based on their derivational relationships (line 2) and initializes a consecutive false alarm counter $K_{counter}$, an empty feedback (observation) trace *obsTrace*, and an unlabeled alarm set U (line 3). Subsequently, the algorithm enters the main loop (line 4), selecting an alarm for user review in each iteration. The algorithm adheres to exploitation instead of exploration under three conditions (lines 5–13). First, at the beginning of the interaction, when no user feedback is available, the system must rely on the initial model (lines 5–6). Second, after the user confirms a true alarm, the system resets the false alarm counter $N_{counter}$ and continues to trust the current model (lines 7–9). Third, even if the user reports a false alarm, the system will continue to exploit as long as the number of consecutive false alarms has not reached the preset threshold $K_{trigger}$ (lines 12–13). Only when the count of consecutive false alarms reaches this threshold does the algorithm determine that the greedy strategy may be stuck in a local optimum and trigger an exploration step (line 15–16). After an alarm is selected, the system obtains user feedback and updates its state for the next iteration (lines 17–19).

5.2 Relevance-Based Alarm Clustering

To facilitate an effective exploration, our first step is to group alarms based on their inter-relevance. Our approach is from the key insight in Section 4.1: the less numerous and proximate the shared root causes between two alarms, the lower their correlation and the lower the overlap in the information obtained from inspecting them. Based on this, we designed a clustering algorithm that defines a “relevance” score by counting nearby shared root causes to measure this correlation and

Algorithm 2 Alarm Clustering via Maximum Relevance

```
Require: A alarm set $A$, a Bayesian network $G$.
Ensure: Clusters of alarms $C = \{C_1, C_2, \dots, C_k\}$
 1: procedure AlarmClustering($A, G$)
    Phase 1: Relevance Computation
 2:     Let $R$ be a map to store relevance scores, initialized to 0.
 3:     for each distinct pair of alarms $(a, b) \in A \times A$ do
 4:         $ancestors_a \leftarrow$ GetAncestors($G.graph, a, n$)          ▷ Detailed in Example 5.1
 5:         $ancestors_b \leftarrow$ GetAncestors($G.graph, b, n$)
 6:         $R(a, b) \leftarrow |ancestors_a \cap ancestors_b|$          ▷ Compute relevance as intersection size
    Phase 2: Iterative Merging
 7:     Initialize a Union-Find structure $UF$ with each alarm $a \in A$ in its own set.
 8:     while $\max(R) > 0$ do
 9:         $maxRelevance \leftarrow \max(R)$
10:         $maxPairs \leftarrow \{(a, b) \mid R(a, b) = maxRelevance\}$          ▷ Find all pairs with max relevance
11:         for each pair $(a, b) \in maxPairs$ do
12:             $UF$.union($a, b$)
13:         $processedAlarms \leftarrow \bigcup_{(a,b) \in maxPairs} \{a, b\}$          ▷ Collect all alarms merged in this step
14:         for each alarm $p \in processedAlarms$ do
15:             for each alarm $x \in A$ do
16:                 $R(p, x) \leftarrow 0$          ▷ Remove processed alarm $p$ from future consideration
17:                 $R(x, p) \leftarrow 0$
18:     return the connected components from $UF$
```

information overlap. The goal of this algorithm is to partition alarms into clusters such that alarms *within* a cluster are highly correlated and have high information overlap, while alarms *between* different clusters are relatively independent and have high information disparity. This partitioning is foundational for the subsequent exploration phase, enabling it to select the alarm that offers the most distinct information based on the current observation trace. Algorithm 2 details this clustering process, which proceeds as follows:

Phase 1: Relevance Computation. The algorithm first computes a relevance score $R(a, b)$ for each distinct pair of alarms (a, b) (lines 2–6). This score is defined as the number of common ancestors within a predefined distance threshold n in the Bayesian network.[5] A concrete computing example is specified in Example 5.1. Setting a threshold n serves two purposes: it ensures that only nearby ancestors with significant influence are considered—since influence decays with distance—and it improves computational efficiency.

Phase 2: Iterative Merging. The algorithm then enters an iterative merging loop. In each iteration, it identifies the alarm pair(s) with the highest current relevance score(s) (lines 9–10). It then uses a Union-Find data structure to merge these most strongly related pairs (lines 11–12) and set the relevance scores involving the alarms in the pairs to 0 (lines 13–17), which makes those alarms unable to merge with other alarms. We merge only the most relevant pairs in each step for two reasons: (1) alarms in a Bayesian network can easily share common ancestors due to high connectivity, and clustering based on any shared ancestry could lead to over-clustering into large, meaningless clusters; (2) conversely, using a fixed relevance threshold might overlook meaningful relationships between less-correlated alarms and imprecisely treat as equal all relevance above the relevance threshold, despite their different intrinsic scores. Therefore, by iteratively merging only the most significant pairs, our algorithm ensures that the resulting clusters are internally

[5]We only take tuples into account when computing distance; ground clauses are ignored.

Algorithm 3 Exploration-Oriented Alarm Ranking

```
Require: Clusters of alarms $C = \{C_1, C_2, \dots, C_k\}$, a set of the unlabeled alarms $U$, a feedback trace $obsTrace$.
Ensure: The selected alarm $X^* \in U$ to be labeled next.
 1: procedure SelectForExploration($C, U, obsTrace$)
 2:     $T \leftarrow \{ a_t \mid \text{alarm } a_t, (a_t, \texttt{true}) \in obsTrace\}$
 3:     $F \leftarrow \{ a_f \mid \text{alarm } a_f, (a_f, \texttt{false}) \in obsTrace\}$
 4:     procedure Key($X$)
 5:         $C_X \leftarrow$ the cluster in $C$ that contains alarm $X$.
 6:         $\bar{C}_X \leftarrow C_X \setminus \{X\}$
 7:         $k_1 \leftarrow [\bar{C}_X \cap T \neq \emptyset]$        ▷ Term 1: Prioritize clusters with no known true alarms.
 8:         $k_2 \leftarrow |\bar{C}_X \cap F|$        ▷ Term 2: Prioritize clusters with fewer known false alarms.
 9:         $k_3 \leftarrow -|\bar{C}_X \cap U|$        ▷ Term 3: Prioritize clusters with more unlabeled alarms.
10:         $k_4 \leftarrow \text{Pr}(X, obsTrace)$        ▷ Term 4: Prioritize alarms with lower posterior probability.
11:         return $(k_1, k_2, k_3, k_4)$        ▷ Return the 4-tuple key for lexicographical comparison.
12:     return $\text{argmin}_{X \in U}$ Key($X$)        ▷ Select the alarm with the smallest key.
```

cohesive and relatively distinct from one another without completely ignoring relatively weaker relationships.

After the loop terminates, the connected components in the Union-Find structure represent the final alarm clusters (line 18). Each cluster comprises a set of alarms that are highly related due to their shared common ancestors, whereas cross-cluster correlations are minimized.

Example 5.1. Take the Bayesian network in Figure 4 as an example. Here we set the threshold n to 1, which means we only take into account the ancestors at distance 1. The direct ancestors of escE(e1) are R_3(e1, v1, h0) and R_3(e1, v1, h1). However, as stated before, we only take tuples into account when computing distance; the two ground clauses do not stop us from continuing the search. After searching up, we have 4 tuples {VH(v1, h0), VH(v1, h1), escH(h0), escH(h1)} at distance 1, 3 tuples {HFH(h10, h0), HFH(h10, h1), escH(h10)} at distance 2, and 1 tuple {FH(h10)} at distance 3. Thus,

$$\text{GetAncestors}(G.graph, \text{escE(e1)}, n) = \{\text{VH(v1, h0)}, \text{VH(v1, h1)}, \text{escH(h0)}, \text{escH(h1)}\}$$

Similarly, we can compute the relevance scores for other alarms:

$$R(\text{escE(e1)}, \text{escE(e2)}) = R(\text{escE(e1)}, \text{escE(e3)}) = 2$$
$$R(\text{escE(e2)}, \text{escE(e3)}) = 4$$
$$\forall a, b \in \{\text{escE(e4)}, \text{escE(e5)}, \text{escE(e6)}, \text{escE(e7)}\}, R(a, b) = 4$$
$$\text{else } R(a, b) = 0 \text{ (Notice that } R(a, b) = R(b, a))$$

Then, starting from the max relevance score 4, we merge escE(e2) and escE(e3) into a cluster, and merge escE(e4), escE(e5), escE(e6) and escE(e7) into another cluster. After that, we set all the relevance scores involving them to 0. Now, all the relevance scores are 0, and we export the connected components in the Union-Find structure, which are:

$$\texttt{Cluster 1} = \{\text{escE(e1)}\},\ \texttt{Cluster 2} = \{\text{escE(e2)}, \text{escE(e3)}\}$$
$$\texttt{Cluster 3} = \{\text{escE(e4)}, \text{escE(e5)}, \text{escE(e6)}, \text{escE(e7)}\}$$

We can see that although escE(e1) has positive relevance scores with escE(e2) and escE(e3), they are not large enough to make them grouped into a cluster.

5.3 Exploration-Oriented Alarm Ranking

During an exploration step, our goal is to find the alarm with the highest expected information gain. Guided by the theoretical analysis and solution in Section 4.2, Algorithm 3 computes a 4-tuple sort key, (k_1, k_2, k_3, k_4), for each uninspected alarm X and uses lexicographical comparison to identify the optimal candidate for exploration. The first three terms are the same as $([t_X > 0], -f_X, u_X)$ from Section 4.2, which estimate the expected information gain by the certainty ($[t_X > 0]$ and $-f_X$) and the uncertainty (u_X) of X's cluster. The last term k_4 serves as a secondary tie-breaker with the least importance, taking effect only when the primary cluster-level metrics (k_1–k_3) yield multiple candidates. It prioritizes the lowest-probability alarm to avoid overlap with the exploitation, which naturally targets high-confidence candidates. This ensures exploration remains orthogonal and focuses on genuinely uncertain regions. Our ablation studies in Section 6.4 confirm that k_4 serves as an effective supplement, providing an additional performance boost, though it is less critical than the primary cluster selection logic which is the fundamental determinant of success.

By sorting these 4-tuples lexicographically to find the minimum, Beer can find the alarm that fits our exploration goals, enabling effective explorations.

Example 5.2. In `Step 5` in Table 1b, keys of the three unlabeled alarms sorted by Key are:

$$\textsc{Key}(\text{escE}(\texttt{e2})) = (0, 0, 1, 0.6730),\ \textsc{Key}(\text{escE}(\texttt{e3})) = (0, 0, 1, 0.6730),\ \textsc{Key}(\text{escE}(\texttt{e7})) = (1, 2, 0, 0.9504)$$

By the sorted result, we can select escE(e2) or escE(e3) for exploration.

6 Experimental Evaluation

Our evaluation will answer the following questions:

RQ1. How effective is Beer in overcoming the local optima problem inherent in the original exploitation-only Bayesian program analysis?
RQ2. How effective is Beer compared to the ϵ-greedy exploration strategy, a representative among existing exploration strategies, which ignores the semantics of the program analysis and the relationship between alarms?
RQ3. What is the individual contribution of each component in Beer to its effectiveness, as revealed by ablation studies?
RQ4. How effective is Beer at enhancing recent approaches that boost Bayesian program analyses in modeling?
RQ5. How sensitive is Beer to the training benchmark selection for tuning the hyperparameter $K_{trigger}$, the exploration trigger number?
RQ6. How scalable is Beer and Beer-enhanced Bayesian program analysis?

We will describe our experimental setup in Section 6.1 and then answer the above research questions in the following sections, respectively.

6.1 Experimental Setup

Instance analyses. We consider three instance analyses in our evaluation: (1) a flow- and context-sensitive Java datarace analysis [Naik et al. 2006] that identifies all possible pairs of statements that may simultaneously access the same heap object, with at least one of the accesses being a write; (2) a flow- and context-insensitive Java thread-escape analysis [Naik et al. 2012] that identifies all possible statements whose operation targets may be accessed by multiple threads; (3) a C taint analysis provided by BayeSmith [Kim et al. 2022] for format-string and integer-overflow errors. Table B.1 in Appendix B presents the statistics of the analyses.

Benchmarks. We select the Java benchmarks for the datarace analysis and the thread-escape analysis from BinGraph [Zhang et al. 2024], and the C benchmarks for the taint analysis from BayeSmith [Kim et al. 2022]. Table B.2 in Appendix B presents the statistics of the benchmarks.

Ground truth. For the datarace analysis, we use the manually inspected ground truth provided by Bingo [Raghothaman et al. 2018]. For the thread-escape analysis, following prior work [Shi et al. 2025; Zhang et al. 2024] and common practice in program analysis studies [Jeon et al. 2018; Li et al. 2018; Mangal et al. 2015; Zhang et al. 2017a], we use the highly precise results produced by a heavy CEGAR-based flow- and context-sensitive analysis [Zhang et al. 2013] as the ground truth. For the taint analysis, we use the ground truth from CVE reports and previous works provided by BayeSmith [Kim et al. 2022].

Baselines. We demonstrate the effectiveness of our Exploration-Exploitation strategy in overcoming the local optima problem by comparing it against the greedy strategy used in the original Bayesian program analysis (denoted by Greedy), which selects the unlabeled alarm with the highest posterior marginal probability at each step. Additionally, we implement the classic ϵ-greedy exploration strategy (denoted by ϵ-Greedy) [Sutton et al. 1998]. At each interaction step, this strategy explores by selecting an alarm uniformly at random with probability ϵ, and otherwise (with probability $1 - \epsilon$) it chooses the alarm with the highest current confidence score. We set ϵ to 0.1, which yields the best performance for the analyses among the candidates $\{0.05, 0.1, 0.3, 0.5\}$, and compare our approach against it to show that our method is more effective at selecting alarms that yield a high gain of new information.

Metrics. Following the past work [Raghothaman et al. 2018; Shi et al. 2025; Zhang et al. 2024], we use three metrics to measure the effectiveness of an interaction strategy in finding true alarms: **inversion (count)**, **Rank-100%-FP**, and **Rank-90%-FP**. The inversion count represents the number of pairs of a false alarm inspected by the user before a true alarm. Formally, it is calculated as $\text{Inversion}(\textit{obsTrace}) = \sum_{t \in T, f \in F} [\text{rank}(f, \textit{obsTrace}) < \text{rank}(t, \textit{obsTrace})]$,[6] where the function "$\text{rank}(a, \textit{obsTrace})$" returns the index of the alarm a in *obsTrace*. For example, the inversion count of the *obsTrace* in Table 1a is 2 because there is one false alarm escE(e5) inspected before two true alarms escE(e2) and escE(e3). Rank-100%-FP is the number of inspected false alarms when all true alarms have been identified, while Rank-90%-FP is the number of inspected false alarms when 90% of true alarms have been identified. Lower values for these metrics signify that a user can access valuable true alarms more quickly, reflecting a better user experience when a user has a budget to check only a small portion of the alarms.

Hyperparameter settings. For the hyperparameter $K_{trigger}$, which defines the number of consecutive "false" alarms required to trigger an exploration step, we perform a linear search to select its optimal value on a training set. We follow the training set selection of Zhang et al. [2024] and use {`jspider,hedc,weblech`} and {`ftp,javasrc-p,weblech`} for the datarace analysis and the thread-escape analysis, respectively. The training set for the taint analysis comprises {`jhead`, `latex2rtf`, `a2ps`, `shntool`}, which are small benchmarks containing no more than 30 total alarms and 6 true alarms. We set $K_{trigger} = 5, 2, 4$ for the datarace analysis, the thread-escape analysis, and the taint analysis, respectively.

For the hyperparameter n, which rules out ancestors whose distances to the given alarm are above it during clustering, we tune it on the same training sets as above, with the objective of balancing the purity of alarm truth within a cluster and the number of clusters. We set $n = 3, 1, 2$ for the datarace analysis, the thread-escape analysis, and the taint analysis, respectively.

[6] [] represents the Iverson Bracket introduced in Section 4.2.

Table 2. Summary of metrics for effectiveness of Beer. "Oracle", "Alarm" and "Clause" in "Analysis info" are the number of true alarms, the total number of alarms reported by the static analysis, and the numbers of ground clauses in the derivation graph after Bingo's graph optimization, respectively. "Average Reduction" represents the average reduction ratio compared to baseline **Greedy**, and "Average Reduction⋆" in the datarace analysis represents the average reduction ratio of the last four large programs.

Program		Analysis info			Inversion			Rank-100%-FP			Rank-90%-FP		
		Oracle	Alarm	Clause	Beer	ϵ-Greedy	Greedy	Beer	ϵ-Greedy	Greedy	Beer	ϵ-Greedy	Greedy
Datarace analysis	`raytracer`	3	187	958	78	75	78	26	25	26	26	25	26
	`luindex`	2	1,236	17,771	25	25	24	13	13	12	13	13	12
	`ftp`	75	571	42,435	4,324	5,130	6,397	113	121	130	73	83	99
	`avrora`	29	1,230	45,810	2,724	4,679	4,032	428	1,026	1,044	271	306	371
	`xalan`	75	2,423	100,888	7,900	16,549	23,845	233	337	401	128	279	400
	`sunflow`	171	2,288	164,847	18,731	22,720	26,816	515	494	565	294	333	422
Average Reduction					26.28%↓	8.22%↓		19.08%↓	5.45%↓		23.87%↓	13.42%↓	
Average Reduction⋆					40.47%↓	12.41%↓		30.71%↓	9.29%↓		37.89%↓	21.25%↓	
Thread-escape analysis	`montecarlo`	54	89	133	285	513	583	24	19	24	7	13	12
	`raytracer`	233	319	556	3,917	4,818	6,884	47	60	86	32	40	61
	`pool`	312	432	598	7,238	9,083	11,199	57	86	93	33	45	88
	`hedc`	287	380	875	3,721	4,144	6,950	56	56	91	31	30	57
	`jspider`	430	624	1,051	14,029	16,555	35,194	124	142	188	58	83	137
	`toba-s`	998	1,278	2,220	33,235	38,763	83,988	280	279	280	53	75	251
Average Reduction					49.44%↓	34.68%↓		26.09%↓	20.31%↓		55.64%↓	38.64%↓	
Taint analysis	`optipng`	1	67	730	17	13	13	17	13	13	17	13	13
	`urjtag`	6	35	920	74	102	96	14	17	16	14	17	16
	`sam2p`	12	20	118	88	92	96	8	8	8	8	8	8
	`autotrace`	18	77	407	123	693	1,062	7	41	59	7	41	59
	`sdop`	65	150	688	556	648	647	13	17	17	13	16	16
Average Reduction					20.59%↓	6.50%↓		18.68%↓	4.85%↓		17.72%↓	4.85%↓	

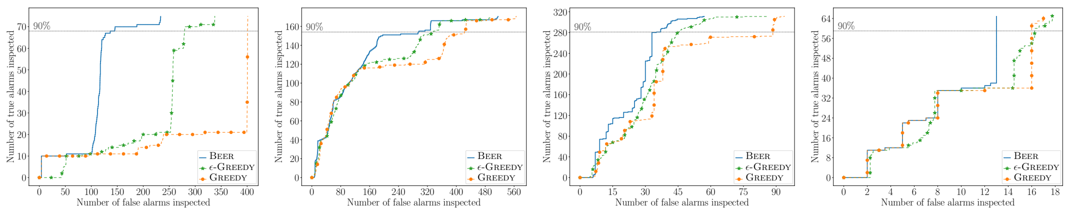
Experiment environment. We conduct the experiments on a Linux machine with 2.40 GHz processors, 256 GB RAM and Nvidia RTX 4090 running Python 3.12.9. We use the Chord framework [Naik et al. 2006] for Java and the Sparrow framework [Oh et al. 2012] for C as the frontend, and the Bingo framework [Raghothaman et al. 2018] for Bayesian program analysis as the backend. For Bayesian inference used in each iteration of Bayesian program analysis to obtain the posterior marginal probability of the unlabeled alarms, we use FastLBP [Feng and Zhang 2025], a GPU-accelerated high-performance implementation of Loopy Belief Propagation (LBP) algorithm for program analysis [Wu et al. 2025].

To ensure the stability and reproducibility of our results, we execute each experiment four times. The performance metrics presented in this paper are the arithmetic mean of these four runs.

6.2 Effectiveness

Beer significantly outperforms the baseline method, the conventional Greedy strategy, across all metrics, with its advantages being particularly pronounced in complex programs which require many rounds of interaction to identify the true alarms. As discussed in Section 2.2, the Greedy strategy is prone to getting stuck in local optima when dealing with highly correlated alarms. Beer effectively mitigates this issue by introducing the Exploration-Exploitation mechanism.

Table 2 presents the detailed experimental results. Beer achieves an average reduction of 49.44% in inversion count, 26.09% in Rank-100%-FP, and 55.64% in Rank-90%-FP for the thread-escape analysis, and 20.59%, 18.68%, and 17.72% for the taint analysis, compared to the Greedy strategy. Similarly, in the datarace analysis, Beer shows significant reductions of 26.28%, 19.08%, and 23.87%, respectively. The performance improvement is especially notable in larger programs which need



(a) *DR*:`xalan` (b) *DR*:`sunflow` (c) *TE*:`pool` (d) *TT*:`sdop`

Fig. 9. The representative average ITCs for the *datarace* (*DR*) analysis, the *thread-escape* (*TE*) analysis, and the *taint* (*TT*) analysis.

numerous rounds of interaction to find all true alarms, such as `ftp`, `avrora`, `xalan`, and `sunflow`, where the average reductions reach 40.47%, 30.71%, and 37.89%.

The average **Interaction Trace Curves (ITCs)** in Figure 9 visualize the interaction trace, which shows Beer's superiority.[7] Each point (x, y) on the curve signifies that the user has identified y true alarms after inspecting x false alarms. The ITC is closely related to inversion count, calculated as $\text{Inversion}(obsTrace) = \int_0^{|T|} f^{-1}(y)\,dy$, which represents the area enclosed by the ITC and the y-axis. The smaller the integration, the smaller the inversion count, and thus the higher the efficiency. That means an ideal curve should be a vertical line, which signifies that a user can discover all true alarms without inspecting any false alarm. As seen in the graphs, Beer's curve (blue) is the closest to the ideal shape in the vast majority of cases, outperforming Greedy's orange dashed line for most of the process. For instance, in the case of `xalan` (Figure 9a), the Greedy strategy gets stuck in a plateau for about 200 rounds of interaction after finding around 10 true alarms, failing to discover new true alarms for an extended period—a typical symptom of the local optimum problem. In contrast, Beer triggers exploration due to the continuous inspection of false alarms during this process. At the mark of approximately 100 inspected false alarms, it successfully breaks out of this dilemma thanks to the prior exploration and begins to efficiently identify new true alarms.

The only three outliers are `raytracer` and `luindex` in the datarace analysis and `optipng` in the taint analysis. The reason is that there are few true alarms and the baseline approach is already able to identify all of them by inspecting a very small fraction of the alarms. As a result, there is little room for improvement. Consequently, all three strategies show nearly identical performance on these benchmarks.

In summary, the results strongly validate that Beer, through its domain-specific exploration strategy, effectively overcomes the local optima problem inherent in original Bayesian program analysis and significantly improves the efficiency of finding true alarms.

6.3 Comparison against ϵ-Greedy

To further demonstrate the effectiveness of our proposed domain-specific exploration strategy, we introduce the classic ϵ-Greedy strategy as another baseline for comparison. The ϵ-Greedy strategy, at each interaction step, triggers exploration with a probability of ϵ which selects an alarm completely at random from all unlabeled alarms and otherwise exploits using the same greedy approach as the Greedy strategy. The experimental results clearly reveal two core insights:

A simple exploration mechanism can outperform a purely greedy strategy. As shown in Table 2, in most test cases, the ϵ-Greedy strategy's performance is superior to the traditional Greedy strategy. For example, in the thread-escape analysis, ϵ-Greedy achieves average reductions of

[7]For the ITCs for all the benchmarks, please refer to Appendix C.

Table 3. Summary of metrics for the ablation experiment. The descriptions in the "setting" column specify the corresponding ablation in exploration step. Statistics are average reduction compared to Greedy.

Setting	**Datarace analysis**			**Thread-escape analysis**			**Taint analysis**		
(Settings are detailed in Section 6.4)	**Inver.**	**R-100%-FP**	**R-90%-FP**	**Inver.**	**R-100%-FP**	**R-90%-FP**	**Inver.**	**R-100%-FP**	**R-90%-FP**
Beer	26.28%↓	19.08%↓	23.87%↓	49.44%↓	26.09%↓	55.64%↓	20.59%↓	18.68%↓	17.72%↓
Setting 1	18.70%↓	16.98%↓	18.09%↓	50.04%↓	32.20%↓	55.56%↓	14.84%↓	11.06%↓	11.36%↓
Setting 2	19.35%↓	10.01%↓	23.70%↓	48.52%↓	25.91%↓	55.06%↓	25.27%↓	22.80%↓	23.09%↓
Setting 3	19.51%↓	11.15%↓	18.96%↓	35.62%↓	30.59%↓	34.82%↓	5.71%↑	9.80%↑	10.39%↑
Setting 4	6.70%↓	9.81%↓	6.48%↓	7.87%↓	15.62%↓	18.52%↓	6.12%↓	3.83%↓	3.69%↓

34.68%, 20.31%, and 38.64% over the Greedy strategy in inversion count, Rank-100%-FP, and Rank-90%-FP, respectively. This result provides strong evidence for the necessity of introducing the Exploration-Exploitation framework into interactive Bayesian program analysis. Even a naive exploration method like ϵ-Greedy, which is simple and disregards any correlation between alarms, can effectively help the analysis escape the dilemma of local optima, thereby improving the efficiency of finding true alarms.

Although the ϵ-Greedy strategy is effective, Beer's domain-specific exploration strategy performs better. Considering inversion counts here, in the thread-escape analysis and the taint analysis, Beer's average performance improvement (49.44% and 20.59%) is much higher than that of ϵ-Greedy (34.68% and 6.50%); in the large-scale programs of the datarace analysis, Beer's advantage is even larger (e.g., a 40.47% reduction compared to just 12.41% for ϵ-Greedy). This performance gap highlights the value of introducing domain knowledge into the exploration strategy. The randomness of ϵ-Greedy means that it might waste precious exploration opportunities on inspecting an alarm with low informational value. In contrast, Beer, through its clustering-based heuristic strategy, can more purposefully select alarms that are most likely to provide new information for exploration, thereby achieving more precise and effective performance gains. The `sunflow` benchmark in the datarace analysis (as shown in Figure 9b) is an exception where ϵ-Greedy performed slightly better on Rank-100%-FP. This suggests that random exploration can occasionally get lucky and select a critical alarm. But in terms of overall average performance, Beer's systematic approach is clearly more reliable and superior.

6.4 Ablation Studies

To decompose the contributions of each component in Beer's exploration strategy, we conducted a series of ablation studies. We designed four variants of Beer, each ablating a part of its core exploration logic. All results are summarized in Table 3.

The four experimental settings correspond to the following strategies:

(1) A random alarm in the best cluster(s) sorted by (k_1, k_2, k_3): This setting retains the logic for selecting the most informative cluster(s) using the sort key (k_1, k_2, k_3) but ablates k_4. Instead of selecting the lowest-probability alarm, it picks one at random from the best cluster(s).
(2) The alarm with the highest probability in the best cluster(s) sorted by (k_1, k_2, k_3): This setting also retains the best-cluster selection logic but inverts the k_4 strategy, choosing the alarm with the highest probability within the cluster.
(3) The alarm with the lowest probability in a random cluster: This setting ablates the cluster selection logic (k_1, k_2, k_3), selecting a cluster at random, but retains the k_4 logic by choosing the lowest-probability alarm within that random cluster.
(4) The alarm with the lowest probability (no clustering): This is the most simplified exploration, completely removing the concept of clustering and directly selecting the alarm with the lowest probability from all uninspected alarms.

Table 4. Summary of metrics for Beer's enhancement upon BinGraph and BayeSmith. "**Base.**" denotes results derived from Bayesian networks learned by contemporary approaches: BinGraph for the datarace and thread-escape analyses, and BayeSmith for the taint analysis. "**Beer**" and "**ϵ-Greedy**" are the results yielded by applying Beer and the ϵ-Greedy strategy to the improved Bayesian networks, respectively. We set $K_{trigger} = 8, 2, 3$ and $n = 2, 1, 2$ for the datarace analysis, the thread-escape analysis, and the taint analysis, respectively. BinGraph's heavy pre-analysis failed on xalan, so we cannot the enhancement on xalan.

Program		**Analysis info**			**Inversion**			**Rank-100%-FP**			**Rank-90%-FP**		
		Oracle	Alarm	Clause	**Beer**	**ϵ-Greedy**	**Base.**	**Beer**	**ϵ-Greedy**	**Base.**	**Beer**	**ϵ-Greedy**	**Base.**
Datarace analysis	`raytracer`	3	187	1,595	19	23	24	7	8	8	7	8	8
	`luindex`	2	976	6,534	29	26	26	14	13	13	14	13	13
	`ftp`	75	525	22,674	817	988	972	16	21	19	14	18	16
	`avrora`	29	1,000	12,059	3,650	4,532	4,591	689	681	656	211	260	252
	`xalan`	—	—	—	—	—	*failed*	—	—	*failed*	—	—	*failed*
	`sunflow`	171	1,561	28,604	14,255	14,152	12,887	239	243	221	203	169	168
Average Reduction					7.02%↓	1.20%↑		1.49%↓	4.86%↑		2.55%↓	3.25%↑	
Thread-escape analysis	`montecarlo`	54	89	125	305	450	412	25	21	24	10	12	10
	`raytracer`	233	319	458	4,489	5,872	6,817	43	49	54	43	42	52
	`pool`	312	432	494	7,563	9,744	9,591	52	63	59	38	48	42
	`hedc`	287	380	872	4,236	4,462	6,106	54	57	86	30	34	45
	`jspider`	430	645	825	12,798	19,526	20,033	140	162	205	53	75	64
	`toba-s`	998	1,278	1,428	39,761	54,124	53,419	280	279	280	62	91	62
Average Reduction					28.93%↓	5.19%↓		16.16%↓	11.67%↓		12.89%↓	9.10%↑	
Taint analysis	`optipng`	1	67	859	3	13	13	3	13	13	3	13	13
	`urjtag`	6	35	965	41	67	66	7	11	11	7	11	11
	`sam2p`	12	20	141	73	86	93	8	8	8	8	8	8
	`autotrace`	18	77	504	376	590	927	37	42	59	37	42	59
	`sdop`	65	150	707	387	597	607	12	16	16	7	14	15
Average Reduction					46.40%↓	8.80%↓		35.11%↓	5.76%↓		40.78%↓	7.10%↓	

From the experimental results, we can draw the following key conclusions:

First, alarm clustering and the cluster selection mechanism are the core of Beer's success. As shown in Table 3, when the cluster selection logic (Setting 3) or the entire clustering concept (Setting 4) is removed, Beer's performance degrades dramatically. Across the three analyses, the average inversion count reduction plummets from 32.10% to 16.47% and 6.90%, respectively. This provides strong evidence that our strategy of clustering based on alarm correlation and prioritizing the most uncertain clusters is the primary contributor to the performance improvement.

Second, the strategy of selecting a low-probability alarm within the best cluster (k_4) is an effective and beneficial supplement. Compared with the standard Beer, the performances of Setting 1 and Setting 2 show degradation. Across the three analyses, the average inversion count reduction for standard Beer is 32.10%, which is higher than 27.86% for the random choice setting (Setting 1) and 31.04% for the highest-probability choice setting (Setting 2). This indicates that after identifying the most valuable region for exploration (i.e., the best cluster), choosing an alarm that counteracts the tendency of the exploitation strategy indeed provides an additional performance boost.

In summary, the ablation studies demonstrate that Beer's primary advantage stems from its ability to "find the right region to explore" (clustering and cluster selection), while the capability to "make the best choice within that region" (the k_4 strategy) serves as an effective supplementary tactic that further optimizes the exploration's impact.

6.5 Enhancement upon Other Contemporary Approaches

Since Beer is an instance of the Exploration-Exploitation paradigm, it is orthogonal and complementary to recent approaches that improve Bayesian network modeling for derivation graphs [Kim

Table 5. Summary of metrics for the leave-one-out cross-validation in the thread-escape analysis. Statistics are average reduction compared to GREEDY.

Settings	Inversion	Rank-100%-FP	Rank-90%-FP
BEER	49.44%↓	26.09%↓	55.64%↓
BEER$_C$	53.34%↓	18.83%↓	58.99%↓

et al. 2022; Shi et al. 2025; Zhang et al. 2024]. To evaluate the enhancement potential of BEER, we select two representative approaches that improve the Bayesian network modeling for the derivation graph: BINGRAPH which improves abstraction selection, and BAYESMITH which improves the probability settings of different use cases derived from a single rule. As shown in Table 4, BEER outperforms BINGRAPH and BAYESMITH across three metrics by average margins of 17.98%, 8.83%, 7.72% and 46.40%, 35.11%, 40.78%, respectively. This demonstrates the effectiveness of using BEER to enhance state-of-the-art approaches. Two outliers, `luindex` and `sunflow`, were observed. For `luindex`, as noted in Section 6.2, the local optima problem does not manifest as the baseline approach is already performing well. For `sunflow`, our analysis reveals that its alarms form many small clusters, indicating they are loosely correlated. As a result, exploration does not yield much benefit, which is not the case for most of the benchmarks (including those in the training set). Conversely, the best-performing `ftp` exhibits a few large clusters (with `avrora` falling in between), indicating that many of its alarms are tightly correlated. We plan to investigate more flexible ways to trigger explorations in the future to avoid useless explorations (as in the case of `sunflow`).

In conclusion, BEER effectively enhances contemporary baselines.

6.6 Sensitivity to Training Data Selection

The performance of the BEER framework depends on a key hyperparameter: the exploration trigger number, $K_{trigger}$. To evaluate the robustness to the training data selection for tuning it, we conducted a leave-one-out cross-validation experiment on the thread-escape analysis, as some benchmarks in the datarace analysis are too large for training.

As shown in Table 5, the performance of BEER under cross-validation (denoted as BEER$_C$) is comparable with, and even slightly better than, the performance of BEER trained on a fixed set. This result indicates that the BEER framework is not sensitive to the selection of training data for tuning $K_{trigger}$; the system can achieve excellent performance consistently with different training sets. This demonstrates the robustness of our method, which maintains high effectiveness across different data subsets.

6.7 Scalability

Table 6 shows that BEER is a lightweight tool with high efficiency and scalability. Its only additional overhead is a one-time alarm clustering step before the interaction begins, which typically takes less time than a single interaction round, making its one-time cost negligible over the many rounds of the entire analysis. In addition, the average time per exploration is roughly the same as the average time per exploitation, and it is dominated by probabilistic inference. Compared to BINGRAPH mentioned in Section 6.5, BEER incurs minimal training costs, introduces no pre-analysis overhead, and scales effectively to large programs. While BINGRAPH's learned abstractions enable faster probabilistic inference, it comes with a substantial training cost (even with parallelization)—20× higher than BEER. Furthermore, its pre-analysis overhead on the last four large datarace benchmarks averages 3,118 seconds, and the process failed entirely on the `xalan` benchmark, demonstrating its limited scalability on large-scale programs, as shown in Table 6.

Table 6. Summary of metrics for scalability of Beer. Tuples and ground clauses are counted from the derivation graph after Bingo's graph optimization. "Train (h)" in Beer and BinGraph represents the total training time respectively. "Cluster (s)" in Beer represents the one-time cost for alarm clustering step before the Bayesian program analysis interaction loop. "Exp. (s)" in Beer represents the average time per exploration. "Pre. (s)" in BinGraph represents its pre-analysis time for finding the suitable abstraction. "Single (s)" represents the average time for every iteration during the interaction for BinGraph the Greedy approach, which consists of only exploitation.

	Program	**#Tuples**	**#Ground clauses**	**Beer**			**BinGraph**			**Greedy**
				Train (h)	**Cluster (s)**	**Exp. (s)**	**Train (h)**	**Pre. (s)**	**Single (s)**	**Single (s)**
Datarace analysis	`raytracer`	907	958	3.56	0.11	1.48	120 (paralleled) [Zhang et al. 2024]	183.78	1.92	1.70
	`luindex`	10,480	17,771		3.67	9.27		1,649.03	2.07	6.68
	`ftp`	27,604	42,435		2.36	15.50		1,447.31	9.22	14.62
	`avrora`	22,413	45,810		4.44	19.46		2,815.44	3.97	19.69
	`xalan`	49,967	100,888		18.25	41.55		*failed*	—	42.50
	`sunflow`	96,319	164,847		26.29	44.20		5,092.96	17.61	44.88
Thread-escape analysis	`montecarlo`	118	133	10.09	0.01	0.05	56 (paralleled) [Zhang et al. 2024]	21.00	0.11	0.07
	`raytracer`	419	556		0.12	0.46		20.99	0.27	0.46
	`pool`	527	598		0.22	0.62		26.88	0.22	0.44
	`hedc`	564	875		0.17	0.13		36.88	0.31	0.33
	`jspider`	825	1,051		0.34	1.07		30.27	0.40	1.62
	`toba-s`	1,564	2,220		1.49	2.59		27.76	0.72	2.32

7 Related Work

Our approach is mostly related to Bayesian program analysis and Exploration-Exploitation approaches including statistical-based and reinforcement learning-based. We summarize related prior work and compare the differences between theirs and our work.

Bayesian Program Analysis. Conventional program analyses typically produce too many false alarms, which makes them impractical for real-world use. Bayesian program analysis is a promising approach to tackle this problem: by modeling the semantics of analysis with a probabilistic model, Bayesian program analysis can systematically assign probabilities to program analysis results, and update the probabilities according to user feedback or other kinds of information. Because of this property, there are many studies about Bayesian program analysis in recent years [Chen et al. 2021; Heo et al. 2019; Kim et al. 2022; Li and Zhang 2025; Mangal et al. 2015; Raghothaman et al. 2018; Zhang et al. 2017b, 2024; Zhang and Zhang 2026]. This body of work can be broadly divided into two main thrusts: incorporating different external information as posterior probabilities and refining the probabilistic model.

A key advantage of Bayesian program analysis is its ability to systematically incorporate diverse sources of information that traditional analysis typically ignores. For example, Heo et al. [2019] notice that the commit history of a program provides meaningful information for program analysis. Therefore, they propose an approach to incorporate different versions of the same program to improve the analysis. Dynamic execution is also useful for analysis, but due to its unsoundness, it is typically ignored in program analysis. By using Bayesian program analysis, Chen et al. [2021] introduce a method to leverage data from dynamic program executions to enhance analysis precision. Similarly, traditional program analysis cannot make use of informal information, such as variable names, which express developer intentions. By encoding informal information as soft evidences in Bayesian learning, Li and Zhang [2025] present a way to systematically utilize informal information to guide the analysis.

Beyond incorporating new types of information, research on Bayesian program analysis also focuses on refining the performance of the underlying Bayesian network.

One line of work aims to learn a better probability assignment of rules. For example, Kim et al. [2022] proposed a fine-grained learning approach that considers different applications of the same Datalog rule so that the probability of a single rule applied in different use cases can be assigned differently. Their approach improves the propagation ability of Bayesian program analysis.

A second, distinct research direction focuses on *abstraction selection*, which is also crucial to the propagation ability of the Bayesian program analysis. Unlike conventional program analysis where a more precise abstraction always produces a better result, an abstraction that is too precise may hinder the *generalization* of user feedback in Bayesian program analysis that prioritizes true bugs over false alarms. Zhang et al. [2024] uses offline learning to learn a strategy to refine the abstraction so that its generalization ability is improved. To better utilize user feedback, Shi et al. [2025] proposed an online approach that refines the abstraction according to certain user feedback. Both these approaches can find abstractions with fewer false alarms and propagate user feedback better so that the user can find true alarms in fewer interactions.

These approaches focus on how to improve the quality of probabilities generated by the Bayesian program analysis, and they all use the greedy paradigm to interact with the Bayesian program analysis. Instead, our approach focuses on the interaction: how to perform explorations with Bayesian program analysis so that the user can get more true alarms in fewer interactions. Therefore, our approach is orthogonal to theirs and can be integrated to further enhance performance.

The Exploration-Exploitation Paradigm. Exploration-Exploitation is a fundamental and important paradigm in fields like reinforcement learning and its various applications [Berger-Tal et al. 2014; Mehlhorn et al. 2015; Sinha 2015]. They can be divided into two main directions: using exploration strategies with mathematical guarantees and using deep reinforcement learning (DRL) to tackle complex problems.

For well-studied approaches like ϵ-greedy and UCB (upper confidence bound) that provide theoretical guarantees, they typically model options/actions as independent of each other [Sutton et al. 1998]. However, for Bayesian program analysis, reports are strongly related to each other and share common causes of imprecision from the analysis. Therefore, these approaches do not fit our situation; thus, we develop a domain-specific exploration strategy to tackle this problem.

For DRL approaches, to solve complex real-world problems like Chess and Go, they also need to consider relations between different yet similar options. Thus, researchers typically use deep neural networks so that they can automatically learn basic patterns that can generalize among situations. By using self-play with an effective exploration strategy, AlphaGo Zero [Silver et al. 2017] beats the top human Go player without learning from any human player. However, DRL approaches typically operate in an "offline" scenario, where exploration occurs only during training to improve the model, and exploitation occurs only during evaluation to maximize performance. However, for our situation, exploration requires feedback from the user, which is costly in practice. So it is impractical to perform too many explorations like DRL approaches do to improve the Bayesian network. Therefore, conventional exploration strategies do not satisfy our requirements.

8 Conclusion

To address the "local optima" problem where purely greedy strategies in traditional Bayesian program analysis can get stuck, we introduce the Exploration-Exploitation paradigm and propose the Beer framework. The framework triggers an "exploration" step after encountering a consecutive number of false alarms. Its core is a domain-specific strategy based on program analysis semantics: it first clusters alarms based on shared root causes, and then prioritizes exploring the most uncertain and informative clusters to maximize expected information gain. We have demonstrated the effectiveness of Beer using a datarace analysis, a thread-escape analysis, and a taint analysis.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback, which greatly improved the quality of this paper. We thank Yuanfeng Shi for his suggestions regarding the theoretical analysis section. This work was sponsored by the National Natural Science Foundation of China (NSFC) under Grant No. W2411051 and Huawei.

Data-Availability Statement

The newest version of the artifact for this paper is available at Zenodo [Lin et al. 2026a] (current version at [Lin et al. 2026b]), which includes all source codes, scripts, and data required to reproduce the results presented in Section 6. Specifically, Table 2, Figure 9, Table 3, Table 4, Table 5, and Table 6 can be reproduced by the artifact. It also includes a reusability guide to encourage further exploration.

References

Oded Berger-Tal, Jonathan Nathan, Ehud Meron, and David Saltz. 2014. The exploration-exploitation dilemma: a multidisciplinary framework. *PloS one* 9, 4 (2014), e95693.

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (2009). 243–262.

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 (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1154–1165. doi:10.1145/3468264.3468626

Haoyu Feng and Xin Zhang. 2025. GPU-Accelerated Loopy Belief Propagation for Program Analysis. doi:10.48550/arXiv.2509.22337 arXiv:2509.22337 [cs].

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 (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 561–575. doi:10.1145/3314221.3314616

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

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

Hyunsu Kim, Mukund Raghothaman, and Kihong Heo. 2022. Learning probabilistic models for static analysis alarms. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1282–1293. doi:10.1145/3510003.3510098

Tianchi Li and Xin Zhang. 2025. Combining Formal and Informal Information in Bayesian Program Analysis via Soft Evidences. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (April 2025), 1774–1801. doi:10.1145/3720508

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

Haoran Lin, Zhenyu Yan, and Xin Zhang. 2026a. *Beer: Interactive Alarm Resolution in Bayesian Program Analysis via Exploration-Exploitation (Paper Artifact)*. doi:10.5281/zenodo.18812846

Haoran Lin, Zhenyu Yan, and Xin Zhang. 2026b. *Beer: Interactive Alarm Resolution in Bayesian Program Analysis via Exploration-Exploitation (Paper Artifact)*. doi:10.5281/zenodo.18812847

Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. 2015. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 462–473. doi:10.1145/2786805.2786851

Katja Mehlhorn, Ben R Newell, Peter M Todd, Michael D Lee, Kate Morgan, Victoria A Braithwaite, Daniel Hausmann, Klaus Fiedler, and Cleotilde Gonzalez. 2015. Unpacking the exploration–exploitation tradeoff: a synthesis of human and animal literatures. *Decision* 2, 3 (2015), 191.

Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. 1999. Loopy belief propagation for approximate inference: an empirical study. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence (UAI'99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 467–475.

Mayur Naik. 2011. Chord: A Versatile Platform for Program Analysis.

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 (PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 308–319. doi:10.1145/1133981.1134018

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 (POPL '12)*. Association for Computing Machinery, New York, NY, USA, 373–386. doi:10.1145/2103656.2103701

Hakjoo Oh, Kihong Heo, Wonchan Lee, Lee Woosuk, and Kwangkeun Yi. 2012. The Sparrow Static Analyzer. https://github.com/ropas/sparrow

Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided program reasoning using Bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 722–735. doi:10.1145/3192366.3192417

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

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, and Adrian Bolton. 2017. Mastering the Game of Go without Human Knowledge. 550, 7676 (2017), 354–359.

Sabyasachi Sinha. 2015. The exploration–exploitation dilemma: a review in the context of managing growth of new ventures. *Vikalpa* 40, 3 (2015), 313–323.

Richard S Sutton, Andrew G Barto, et al. 1998. *Reinforcement learning: An introduction*. Vol. 1. MIT press Cambridge.

Yiqian Wu, Yifan Chen, Yingfei Xiong, and Xin Zhang. 2025. Belief Propagation with Local Structure and Its Applications in Program Analysis. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering, ASE*, Vol. 25.

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

Xin Zhang, Mayur Naik, and Hongseok Yang. 2013. Finding optimum abstractions in parametric dataflow analysis. *SIGPLAN Not.* 48, 6 (June 2013), 365–376. doi:10.1145/2499370.2462185

Xin Zhang, Xujie Si, and Mayur Naik. 2017b. Combining the logical and the probabilistic in program analysis. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2017)*. Association for Computing Machinery, New York, NY, USA, 27–34. doi:10.1145/3088525.3088563

Yifan Zhang, Yuanfeng Shi, and Xin Zhang. 2024. Learning Abstraction Selection for Bayesian Program Analysis. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (April 2024), 954–982. doi:10.1145/3649845

Yifan Zhang and Xin Zhang. 2026. Fuzzing Guided by Bayesian Program Analysis. *Proc. ACM Program. Lang.* 10, POPL, Article 17 (Jan. 2026), 31 pages. doi:10.1145/3776659

Received 2025-10-09; accepted 2026-02-17