

LLM-Based Alarm Resolution Guided by Bayesian Program Analysis

YIFAN ZHANG, Peking University, China
YUANFENG SHI, Peking University, China
HAORAN LIN, Peking University, China
YINGFEI XIONG, Peking University, China
XIN ZHANG*, Peking University, China

Abstract-interpretation-based static analyzers often report large numbers of alarms due to over-approximation. Although large language models (LLMs) can help filter alarms, per-alarm prompting is often inaccurate and expensive. LLMs often misjudge such end alarms, and the repeated context across queries wastes many tokens. We shift LLM judgment from end alarms to intermediate facts (e.g., alias or flow edges), which are easier to validate. If a fact is judged false, all dependent facts and alarms can be pruned. We capture these dependencies in a derivation graph, enabling analyzer-agnostic pruning for any tool that exposes derivations. Under a token budget, we define the fact impact prioritization problem, which asks which facts to query first to maximize expected downstream pruning. We solve it with Bayesian program analysis by estimating each fact's pruning impact from rule probabilities and fact posteriors. Building on these ideas, we present an LLM-based alarm resolution framework guided by Bayesian program analysis. It iteratively queries high-impact facts that LLMs can judge accurately, prunes downstream nodes when a fact is false, and feeds the judgments back to the Bayesian model as high-confidence evidence. We evaluate our approach on a Java datarace analysis and a C taint analysis, showing that it improves alarm-resolution quality while substantially reducing token consumption compared with both unfiltered static analysis and per-alarm LLM judging.

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

Additional Key Words and Phrases: Static Analysis, Bayesian Networks, Large Language Models

ACM Reference Format:
Yifan Zhang, Yuanfeng Shi, Haoran Lin, Yingfei Xiong, and Xin Zhang. 2026. LLM-Based Alarm Resolution Guided by Bayesian Program Analysis. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 404 (October 2026), 27 pages. https://doi.org/10.1145/3839536

1 Introduction

Abstract-interpretation-based static analyses [22] are widely used to find bugs and security problems. However, they typically rely on over-approximation, which often leads to many false alarms. In

*Corresponding author.

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



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/10-ART404
https://doi.org/10.1145/3839536

practice, a static analysis tool may produce hundreds or thousands of alarms. Manually triaging them is time-consuming, degrades user experience, and often hinders real-world adoption.

Large language models (LLMs) have recently shown impressive capabilities in code understanding and reasoning, which suggests using an LLM to filter alarms from static analysis. A straightforward approach is to ask the LLM to judge each alarm as true or false. However, this one-alarm-at-a-time method has two serious problems. First, it is **inaccurate**: the LLM often misjudges whether an alarm is real, discarding true alarms while retaining false ones. Second, it is **expensive**: when there are many alarms, querying the LLM for each alarm consumes many tokens and repeatedly pays for similar context.

This paper proposes a general approach that avoids these two problems by changing what we ask the LLM to judge. Our key idea is to target the root causes of false alarms: many false alarms are derived from the same incorrect intermediate fact (e.g., an alias relation or a flow edge) introduced by approximation. Therefore, instead of judging alarms one by one, we ask the LLM to judge intermediate facts; once a fact is judged incorrect, we can remove it and also remove many later facts (including many alarms) that depend on it, thereby pruning many false alarms with a single query. Meanwhile, to keep such pruning trustworthy, we only query facts from classes that the LLM can judge with **high accuracy**, which are identified in advance. We capture this structure using a derivation graph built from the analysis process: each node is a derived fact (alarms are also nodes), and each directed edge means that one fact was used to derive another. When an intermediate fact is false, all downstream nodes that depend on it lose their support and can be pruned. This makes our approach analysis-agnostic: any static analysis that can expose or record its derivations can be handled in the same way.

After expanding the LLM's targets from alarms to intermediate facts, the key question becomes: which facts should be checked first under a limited token budget? We call this the *fact impact prioritization problem*. We define our objective using the number of reachable pairs (f, a), where f is an incorrect fact and a is a false alarm that is directly or indirectly derived from f; reducing these pairs quickly leads to pruning many false alarms. Accordingly, we define a fact's *impact* as the expected number of such reachable pairs that would be removed if the fact were pruned (when it is false), and we prioritize facts with higher impact to reduce the number of LLM queries and thus **lower token consumption**. To estimate impact, we leverage Bayesian program analysis [21], which attaches probabilities to derivation rules to model approximation and turns the analysis derivation into a Bayesian model; by incorporating posterior information [5, 8, 12, 21, 28], it computes the probability that each fact in the derivation graph holds. We use these probabilities to estimate each fact's impact and solve the fact impact prioritization problem.

Based on these ideas, we present BayPruner, an LLM-based alarm resolution framework guided by Bayesian program analysis. BayPruner can take arbitrary static analyses as input, automatically extracts the analysis derivation process, and allows users to specify which classes of derived facts can be judged by the LLM with high accuracy (obtained via offline sampling or expert annotation). BayPruner then runs an iterative procedure over these facts. In each iteration, BayPruner uses Bayesian program analysis to estimate the expected impact of candidate facts and selects the one with the highest impact. It then asks the LLM to judge that fact; if the fact is false, BayPruner removes it and all downstream facts that depend on it. Since BayPruner assumes these fact judgments are sufficiently accurate, it feeds the judgment (whether the fact holds) back to the Bayesian model as high-confidence posterior evidence, updates the probability distribution, and selects the next fact to continue the process. Finally, BayPruner stops when querying facts is no longer expected to have higher impact than directly querying alarms, so it does not fall back to the naïve one-alarm-at-a-time method.

```
1  int id(int i){
2      return i;
3  }
4
5  int main(){
6      t = input();
7      t = id(t);
8      a = safe_value();
9      a = id(a);
10     b = input();
11     // ----- BEGIN: huge propagation chain -----
12     X_1 = a & 998244353;
13     X_2 = b ^ 998244535;
14     X_3 = X_1 << 10;
15     // ...
16     X_1500 = X_1300 > X_1400 ? X_1498 : X_1499;
17     // ...
18     X_12000 = X_11995 >> 2;
19     // ----- END: huge propagation chain -----
20     /*
21      * After a sequence of complex computations and
22      * propagations, the data-flow relations are:
23      *   X_11996, X_11997, and X_11998 depends on a
24      *   X_11999 and X_12000 depends on b
25      */
26     cal(X_11996, X_11997, X_11998,
27         X_11999, X_12000);
28 }
29
30
31 void cal(int x1, int x2, int x3, int y1, int y2){
32     int A[100];
33     A[x1] = 1;            // A1
34     A[x2] = 2;            // A2
35     A[x3] = 3;            // A3
36     A[y1 + y2] = 4;       // A4
37     A[y1 - y2] = 5;       // A5
38 }
```

Fig. 1. An example C code fragment.

We evaluate BayPruner on two representative analyses to demonstrate its effectiveness and generality: a datarace analysis on a suite of 8 Java programs ranging from 292 to 529 KLOC, and a taint analysis on a suite of 9 C programs ranging from 5 to 64 KLOC. Compared with (1) running static analysis without LLM filtering and (2) asking the LLM to judge each alarm, BayPruner improves the average F1-score by 4.4× and 2.3×, respectively, and reduces average token usage by 76.6% compared with the latter one-alarm-at-a-time baseline.

Contributions. This paper makes the following contributions:

(1) We improve LLM-based alarm resolution by expanding LLM judgments from alarms to intermediate derived facts: pruning an incorrect fact can eliminate many false alarms at once, and queries are restricted to facts that LLMs can judge with high accuracy.
(2) We define the fact impact prioritization problem and use Bayesian program analysis to estimate fact impact, enabling us to prune more false alarms with fewer tokens.
(3) We propose BayPruner, an LLM-based alarm resolution framework guided by Bayesian program analysis. We implement BayPruner and validate the effectiveness and generality of our approach on two representative analyses.

2 Motivating Example

This section uses the code fragment in Figure 1 as an example to illustrate, in the setting of a taint analysis for detecting memory errors, (i) the imprecision and high token consumption introduced by a naïve one-alarm-at-a-time strategy when using an LLM for filtering, and (ii) how our method effectively addresses both issues.

In this code, variables `t`, `a`, and `b` are assigned at Line 6, Line 8, and Line 10, respectively. The values of `t` and `b` originate from external inputs and may therefore be attacker-controlled, potentially leading to subsequent errors, whereas `a` is assigned a safe value and will not cause harmful effects during later execution. Next, at Line 7 and Line 9, variables `t` and `a` are passed as arguments to the identity function `id`. This transformation does not change the concrete values of these variables; however, for a context-insensitive static analysis, it may introduce spurious over-approximated

Input relations	
$\mathsf{Input}(v)$:	The value of variable v comes from external input.
$\mathsf{Flow}(v_1, v_2)$:	There exists a data flow from v_1 to v_2.
$\mathsf{Memory}(v, s)$:	v involves in memory operations in statement s.
Output relations	
$\mathsf{Taint}(v)$:	Variable v is tainted.
$\mathsf{Alarm}(s)$:	Statement s leads to a memory error.
Derivation rules	
R_1 :	$\mathsf{Taint}(v)$:- $\mathsf{Input}(v)$.
R_2 :	$\mathsf{Taint}(v_2)$:- $\mathsf{Taint}(v_1), \mathsf{Flow}(v_1, v_2)$.
R_3 :	$\mathsf{Alarm}(s)$:- $\mathsf{Taint}(v), \mathsf{Memory}(v, s)$.

Fig. 2. A taint analysis in Datalog.

facts. We return to this issue later. Afterwards, `a` and `b` underwent a complex huge propagation chain involving 12,000 variables from Line 11 to Line 19. The final data-flow relationships are as follows: variable `a` influences the values of `X_11996`, `X_11997`, and `X_11998`, while variable `b` influences `X_11999` and `X_12000`. Next, these five variables were passed as arguments to the function `cal` at Line 27. We assume that this computation is sufficiently complex such that, although a sound static analysis can still recover the data flow due to its sound over-approximation, it is difficult for an LLM or a code agent to reliably infer `a` and `b` from `x1`, `x2`, `x3`, `y1`, and `y2`, because doing so requires a long and complex chain of reasoning. Finally, at Line 33–Line 37, these five variables are used as array indices in memory accesses. Since `a` is safe, `x1`, `x2`, and `x3` remain within valid bounds and do not trigger memory errors. In contrast, `b` is controlled by external input, so `y1` and `y2` may fall outside the normal range, thereby causing the corresponding memory errors. In the following, we describe how a taint analysis for memory-error detection, combined with LLM-based filtering, can be used to ultimately identify these two vulnerabilities.

2.1 A Taint Analysis for Detecting Memory Errors

We employ a taint analysis to produce alarms for the program under analysis. These alarms are subsequently passed to the LLM, which is responsible for filtering them.

Figure 2 presents our taint analysis encoded as a set of Datalog rules. The analysis is designed to be context-insensitive while remaining flow-sensitive. It conservatively treats every program input as a taint source and propagates taint information along all potential data-flow paths. A value is regarded as tainted if it can be directly influenced by an attacker through external input. Whenever such a tainted value is involved in a memory-related operation (for example, array indexing or pointer arithmetic), the analysis issues an alarm.

In addition, Figure 2 provides the complete specification of the input and output relations that the analysis relies on. The analysis is built from three derivation rules: R_1 marks inputs as tainted, R_2 propagates taint through the program, and R_3 captures memory operations that depend on tainted values. All three rules are intentionally over-approximating and may therefore produce spurious derived facts. More concretely, R_1 and R_3 do not account for the range restrictions imposed by conditional branches, whereas R_2 abstracts away differences between calling contexts.

Given a set of input tuples, a Datalog engine repeatedly applies these derivation rules and iterates the evaluation until it reaches a fixed point, that is, a state in which no further output tuples can be derived.

We visualize the Datalog derivation as a directed graph for clarity, and refer to this graph as the *derivation graph*. Figure 3 shows the derivation graph obtained by instantiating the rules in Figure 2

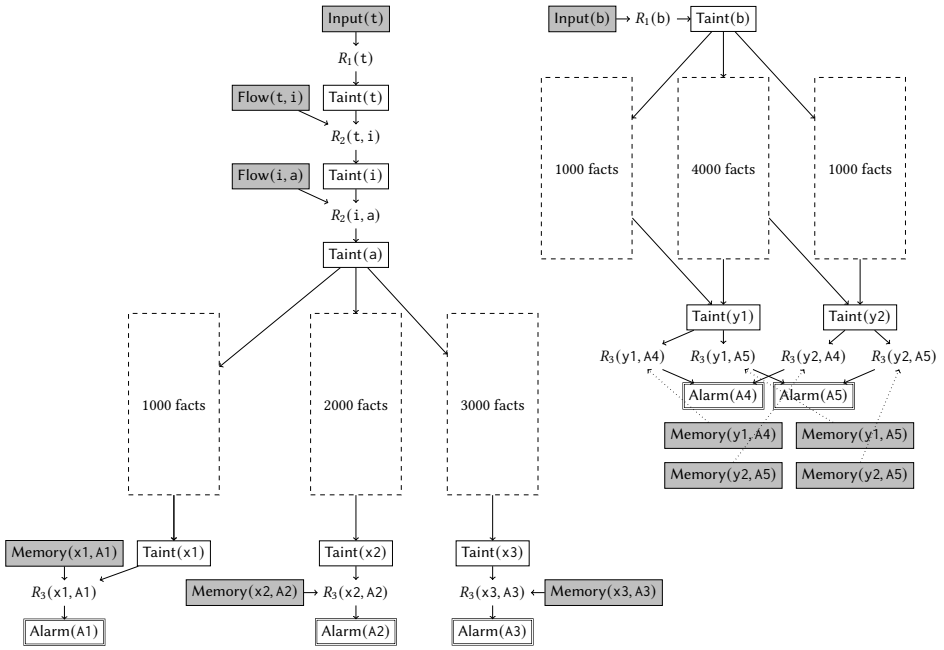


Fig. 3. The derivation graph of the analysis in Figure 2 applied to the code fragment in Figure 1. Vertices with a gray bordered background represent input tuples. Vertices with a white bordered background represent output tuples, while vertices with double white borders represent alarm tuples. Vertices without borders represent rule instances. Large dashed boxes denote omitted portions of the derivation graph, with the number of contained facts annotated. Dashed arrows are used only for visual clarity and have the same meaning as solid arrows.

on the program in Figure 1. The graph contains input tuples (grey nodes with borders), output tuples (white nodes with borders), and rule instances (nodes without borders) that generate the output tuples. Each tuple corresponds to a single fact: input tuples encode the initial facts provided to the analysis, whereas output tuples represent the derived facts. To compactly represent the parts of the derivation that correspond to the complex numerical computations hidden in the code, we use dashed boxes to collapse these subgraphs and annotate each box with the number of facts it contains.

A rule instance corresponds to one concrete application of a derivation rule that produces a new output tuple. For example, the instance $R_2(\texttt{t}, \texttt{i})$ uses Taint(t) together with Flow(t, i) to derive Taint(i). In particular, vertices drawn with double borders and white backgrounds denote alarm tuples. Overall, the analysis reports five alarms: A1, A2, and A3 are false positives, whereas A4 and A5 are true positives. The false alarms arise because the taint analysis is context-insensitive and therefore conflates the two calls to the identity function id. As a result, taint from the attacker-controlled variable t is incorrectly propagated to the safe variable a, which then further contaminates x1, x2, and x3. This mistaken propagation ultimately leads to the alarms reported on the subsequent memory accesses being false.

Now suppose we have a code agent that can be executed directly in the target repository, and we query it separately to decide whether each of the five alarms is real. This naïve approach introduces two issues. (1) To accurately validate an alarm, the agent must invoke tools and reason about the root cause that determines whether the alarm is feasible (i.e., the concrete values or effective

constraints of a and b). However, as assumed above, the agent can hardly complete this reasoning reliably, and is thus prone to misjudging these alarms. (2) The derivation graph makes it clear that different alarms are strongly correlated: A1, A2, and A3 share the same origin, and similarly A4 and A5 form another group. If the agent is asked about each alarm in isolation, it is forced to repeatedly spend context on essentially the same backtracking task (i.e., tracing each alarm back to a and b), which wastes tokens on redundant reasoning.

Several straightforward remedies also fail to adequately address the problem. For example, providing the full derivation graph to the agent is unlikely to help: in practice, static-analysis derivation graphs are often highly entangled and can be substantially larger than the program itself, making it unrealistic and unreliable to expect the agent to discover inter-alarm relationships from such a large structure. Likewise, forcing all alarms to share a single global context is ineffective. Static analyses typically emit many alarms; a small number of queries can quickly exhaust the context budget, and once the context must be compacted, critical information is easily lost, making it difficult to reliably reuse previously established facts.

Next, we demonstrate how BayPruner resolves these challenges.

2.2 Core Idea of BayPruner

The core idea of BayPruner is twofold. (1) Instead of querying the code agent only about alarms, we **extend the query targets to intermediate derived facts**. Incorrect intermediate facts are often the **shared root causes** of many false alarms; by pruning them, BayPruner can indirectly eliminate the corresponding false alarms. To keep such pruning trustworthy, BayPruner only queries facts that the code agent can judge with **high accuracy**. (2) We define the fact impact prioritization problem and systematically address it using Bayesian program analysis. A high-impact fact is one that is **expected to prune more false alarms once queried**. Bayesian program analysis provides a principled way to **compute the probability that each fact holds**. By selecting the fact with the largest impact, BayPruner can **eliminate more false alarms with fewer queries**, thereby reducing token consumption. Next, we use the above example to illustrate these two aspects of BayPruner in detail.

BayPruner expands the validation target of code agents from alarms to intermediate facts. In our example, checking whether Taint(t) and Taint(a) hold is substantially easier than determining whether Alarm(A1) is a true alarm. When a derived fact is judged to be invalid by a code agent, we remove that fact together with all downstream derivations that depend on it, thereby pruning false alarms indirectly. For instance, if Taint(x1) is deemed false, then Alarm(A1) will also be eliminated.

BayPruner allows users to assign a prior accuracy, **indicating whether code agents can correctly judge each derived fact**, and only facts whose prior accuracy is no lower than a threshold are eligible for code-agent validation. Such priors can be obtained through offline sampling or expert annotation. For simplicity, we assume that all derived facts of type Taint have prior accuracy 0.9, whereas all derived facts of type Alarm have prior accuracy 0.5. We set the threshold to 0.9, thereby ensuring that the code agent is queried only on facts for which it is expected to be highly accurate.

The remaining question is how to select derived facts so as to prune as many false alarms as possible with as few queries as possible. Because an alarm may be implied by multiple facts (e.g., Taint(y1) and Taint(y2) jointly derive Alarm(A4)), the optimal solution is, in general, a *set* of derived facts. Consequently, the problem is NP-complete. To avoid overly complex optimization, we adopt a greedy strategy.

Since removing an incorrect fact may not immediately eliminate any false alarm, we therefore introduce a new metric that captures the influence of pruning derived facts on false alarms. Specifically, we aim to reduce the number of pairs (f, a), where f is an incorrect fact and a is a false alarm

derived from f. Once all incorrect facts that derive a false alarm have been pruned, that false alarm is removed; thus, fewer such pairs implies fewer remaining false alarms. In practice, an alarm may be derived from many facts, yet pruning it still removes only a single alarm; naïvely counting pairs would therefore over-credit alarms with many derivations. To mitigate this effect, for an alarm that has n deriving facts, we assign each pair a weight of $\frac{1}{n}$.

We define the *impact* of a fact as the expected reduction in the total weight of these pairs after querying the code agent about that fact. Our greedy procedure then repeatedly selects, under the current state, the fact with the maximum impact to query next. This reduces our task to efficiently identifying the fact with the highest impact, which we call the *fact impact prioritization problem*. BayPruner addresses this problem using Bayesian program analysis.

Bayesian program analysis associates probabilities with derivation rules to quantify the degree of approximation. In our running example, since R_1, R_2, and R_3 are over-approximating, we attach probabilities to their derivations to **indicate how likely a derivation is correct**. For simplicity, we assign each rule a probability of 0.9.

Under this probabilistic interpretation, the derivation graph can be viewed as a Bayesian network [11]. Concretely, each tuple or rule instance t is mapped to a Bernoulli random variable x_t, indicating whether the corresponding fact (or rule application) holds. For instance, the tuple $\mathtt{Taint(t)}$ and the rule instance $R_2(\mathtt{t}, \mathtt{i})$ correspond to the variables $x_{\mathtt{Taint(t)}}$ and $x_{R_2(\mathtt{t},\mathtt{i})}$, respectively.

The dependency structure among these variables follows the edges of the derivation graph. We encode this structure as conditional probability distributions (CPDs) defined per rule instance. For example, we instantiate the conditional probabilities for the rule instances $R_1(\mathtt{t})$, $R_2(\mathtt{t}, \mathtt{i})$, as well as $R_3(\mathtt{y1}, \mathtt{A4})$ and $R_3(\mathtt{y2}, \mathtt{A4})$, as follows:

(1) **For rule instance $R_1(\mathtt{t})$:**

$$\Pr(x_{R_1(\mathtt{t})} \mid x_{\mathsf{Input(t)}}) = 0.9 \quad \Pr(x_{R_1(\mathtt{t})} \mid \neg x_{\mathsf{Input(t)}}) = 0$$
$$\Pr(x_{\mathsf{Taint(t)}} \mid x_{R_1(\mathtt{t})}) = 1 \quad \Pr(x_{\mathsf{Taint(t)}} \mid \neg x_{R_1(\mathtt{t})}) = 0$$

(2) **For rule instance $R_2(\mathtt{t}, \mathtt{i})$:**

$$\Pr(x_{R_2(\mathtt{t},\mathtt{i})} \mid x_{\mathsf{Taint(t)}} \wedge x_{\mathsf{Flow(t,i)}}) = 0.9 \quad \Pr(x_{R_2(\mathtt{t},\mathtt{i})} \mid \neg x_{\mathsf{Taint(t)}} \vee \neg x_{\mathsf{Flow(t,i)}}) = 0$$
$$\Pr(x_{\mathsf{Taint(i)}} \mid x_{R_2(\mathtt{t},\mathtt{i})}) = 1 \quad \Pr(x_{\mathsf{Taint(i)}} \mid \neg x_{R_2(\mathtt{t},\mathtt{i})}) = 0$$

(3) **For rule instances $R_3(\mathtt{y1}, \mathtt{A4})$ and $R_3(\mathtt{y2}, \mathtt{A4})$:**

$$\Pr(x_{R_3(\mathtt{y1},\mathtt{A4})} \mid x_{\mathsf{Taint(y1)}} \wedge x_{\mathsf{Memory(y1,A4)}}) = 0.9$$
$$\Pr(x_{R_3(\mathtt{y1},\mathtt{A4})} \mid \neg x_{\mathsf{Taint(y1)}} \vee \neg x_{\mathsf{Memory(y1,A4)}}) = 0$$
$$\Pr(x_{R_3(\mathtt{y2},\mathtt{A4})} \mid x_{\mathsf{Taint(y2)}} \wedge x_{\mathsf{Memory(y2,A4)}}) = 0.9$$
$$\Pr(x_{R_3(\mathtt{y2},\mathtt{A4})} \mid \neg x_{\mathsf{Taint(y2)}} \vee \neg x_{\mathsf{Memory(y2,A4)}}) = 0$$
$$\Pr(x_{\mathsf{Alarm(A4)}} \mid x_{R_3(\mathtt{y1},\mathtt{A4})} \vee x_{R_3(\mathtt{y2},\mathtt{A4})}) = 1$$
$$\Pr(x_{\mathsf{Alarm(A4)}} \mid \neg x_{R_3(\mathtt{y1},\mathtt{A4})} \wedge \neg x_{R_3(\mathtt{y2},\mathtt{A4})}) = 0$$

All other rule instances are assigned CPDs in the same manner. In this way, the resulting conditional probability distributions connect the Bernoulli variables through semantics-grounded probabilistic relationships.

We compute the probability of each derived fact being true by performing marginal inference [15] on the Bayesian network constructed from Figure 3. The resulting probabilities are shown in the **Rank** column of Table 1a.

Based on the predicted probability of each derived fact being true in the Bayesian network, we define the impact of each fact according to the following formula:

$$\textsc{Impact}(t) = \big(1 - \Pr(x_t)\big) \cdot \textsc{Acc}(t) \cdot \textsc{Weight}(t).$$

Here, $1 - \Pr(x_t)$ denotes the probability that fact t does not hold, $\textsc{Acc}(t)$ denotes the accuracy of the code agent in determining whether fact t holds, and $\textsc{Weight}(t)$ represents the expected reduction in the weighted sum of false-fact–reachable-alarm pairs if t is an incorrect derived tuple and is removed.

The intuition behind this formula is as follows. Only when (1) the fact is indeed false and (2) the code agent correctly identifies it as false can the weighted sum of pairs be reduced by $\textsc{Weight}(t)$. Otherwise, the weighted sum remains unchanged. Therefore, the expected reduction equals the probability of this event occurring multiplied by the reduction achieved once it occurs. In Table 1a, we report the Acc and Weight of each fact, compute their corresponding Impact values, and rank them in descending order.

We now illustrate how Weight is computed using several examples:

(1) $\textsc{Weight}(\mathsf{Taint(x1)}) = 1004 \times \frac{1}{1004} \Pr(\neg x_{\mathsf{Alarm(A1)}} \mid \neg x_{\mathsf{Taint(x1)}}) = 1$. This is because Taint(x1) is a mandatory node for deriving Alarm(A1). If this fact does not hold, Alarm(A1) can no longer be derived, and thus no pairs can reach Alarm(A1). Originally, there are 1004 facts that can reach this alarm, each assigned a weight of $\frac{1}{1004}$. Hence, the total reduction in weighted sum is 1. For all derived facts except Taint(y1) and Taint(y2), the computation of their impact follows the same procedure.

(2) $\textsc{Weight}(\mathsf{Taint(y1)}) = \frac{1001}{6003} \times (\Pr(\neg x_{\mathsf{Alarm(A4)}} \mid \neg x_{\mathsf{Taint(y1)}}) + \Pr(\neg x_{\mathsf{Alarm(A5)}} \mid \neg x_{\mathsf{Taint(y1)}}))$. For Alarm(A4) and Alarm(A5), if the fact Taint(y1) does not hold, both the fact itself and the 1000 facts inside the left dashed box become unreachable from these two alarms. For each alarm, every pair has weight $\frac{1}{6003}$. Since in our definition of a pair (f, a) the alarm a is assumed to be a false alarm, we need to multiply by the probability that Alarm(A4) and Alarm(A5) are false. We assume that $\Pr(\neg x_{\mathsf{Alarm(A4)}} \mid \neg x_{\mathsf{Taint(y1)}}) = \Pr(\neg x_{\mathsf{Alarm(A5)}} \mid \neg x_{\mathsf{Taint(y1)}}) = 0.71$, and thus the total expected reduction in weight is $2 \times 1001 \times \frac{1}{6003} \times 0.71 \approx 0.237$. Similarly, for Taint(y2), suppose that $\Pr(\neg x_{\mathsf{Alarm(A4)}} \mid \neg x_{\mathsf{Taint(y2)}}) = \Pr(\neg x_{\mathsf{Alarm(A5)}} \mid \neg x_{\mathsf{Taint(y2)}}) = 0.70$, then the total expected reduction in weight is $2 \times 1001 \times \frac{1}{6003} \times 0.70 \approx 0.233$.

With this example, we have illustrated the core idea of BayPruner. Next, we provide a complete description of how BayPruner performs alarm resolution more effectively.

2.3 Workflow of BayPruner

The workflow of BayPruner is structured into the following systematic phases:

(1) **Impact Prediction and Selection:** First, BayPruner identifies the derived fact with the maximum impact by leveraging the Bayesian network, the derivation graph structure, and the predicted accuracy of the code agents. This computation is **systematically grounded in analysis semantics**, ensuring the validity of the prioritization. Among the facts for which the code agent's accuracy exceeds a predefined threshold, BayPruner submits the one with the highest impact to the code agent for verification.

(2) **Pruning and Adaptive Feedback:** If the code agent classifies a fact as false, BayPruner prunes that fact along with all its subsequent derived facts in the graph. Regardless of the verification outcome, the result is fed back into the Bayesian network. Given the high reliability of code agents, this result serves as high-confidence posterior information, enabling the Bayesian network to perform probabilistic learning and update its distributions. This systematic feedback loop ensures the **self-adaptivity** of BayPruner.

Table 1. Statistics for each derived fact in Figure 3: its probability of holding, the expected sum of pair weights reduced if it is pruned, the code agent's accuracy on that fact, and the resulting impact. Results are shown before and after incorporating the code agent's feedback on Taint(a) and Taint(b).

(a) $\Pr(x_t)$

Fact	Prob.	Weight	Acc	Impact
Taint(a)	0.72	3	0.9	0.756
Taint(i)	0.81	3	0.9	0.513
Taint(x3)	0.66	1	0.9	0.306
Taint(x2)	0.68	1	0.9	0.288
Taint(x1)	0.7	1	0.9	0.27
Taint(t)	0.9	3	0.9	0.27
Alarm(A3)	0.594	1	0.5	0.203
Alarm(A2)	0.612	1	0.5	0.194
Alarm(A1)	0.63	1	0.5	0.185
Taint(b)	0.9	2	0.9	0.18
Taint(y2)	0.4	0.233	0.9	0.126
Taint(y1)	0.5	0.237	0.9	0.107
Alarm(A4)	0.8	1	0.5	0.1
Alarm(A5)	0.8	1	0.5	0.1

(b) $\Pr(x_t \mid \neg x_{\text{Taint(a)}})$

Fact	Prob.	Weight	Acc	Impact
Taint(b)	0.9	2	0.9	0.18
Taint(y2)	0.4	0.233	0.9	0.126
Taint(y1)	0.5	0.237	0.9	0.107
Alarm(A4)	0.8	1	0.5	0.1
Alarm(A5)	0.8	1	0.5	0.1
Taint(a)	0	-	-	0
Taint(i)	0	-	-	0
Taint(x3)	0	-	-	0
Taint(x2)	0	-	-	0
Taint(x1)	0	-	-	0
Taint(t)	0	-	-	0
Alarm(A3)	0	-	-	0
Alarm(A2)	0	-	-	0
Alarm(A1)	0	-	-	0

(c) $\Pr(x_t \mid \neg x_{\text{Taint(a)}} \wedge x_{\text{Taint(b)}})$

Fact	Prob.	Weight	Acc	Impact
Alarm(A4)	0.9	1	0.5	0.05
Alarm(A5)	0.9	1	0.5	0.05
Taint(y2)	0.87	0.167	0.9	0.02
Taint(y1)	0.88	0.173	0.9	0.019
Taint(b)	1	-	-	0
Taint(a)	0	-	-	0
Taint(i)	0	-	-	0
Taint(x3)	0	-	-	0
Taint(x2)	0	-	-	0
Taint(x1)	0	-	-	0
Taint(t)	0	-	-	0
Alarm(A3)	0	-	-	0
Alarm(A2)	0	-	-	0
Alarm(A1)	0	-	-	0

(3) **Re-computation and Termination:** The impact of the remaining facts is then re-evaluated. If the fact with the highest impact is an alarm, further iterations would cause the marginal utility to degenerate into the naïve one-alarm-at-a-time approach. Consequently, BayPruner terminates the filtering process at this stage. The remaining unpruned alarms constitute the final filtered output. This termination mechanism ensures that BayPruner maintains **non-degenerate efficiency gains**. Otherwise, BayPruner returns to step (1) and continues the process.

Figure 4 illustrates the overall workflow of BayPruner in our example. In the following, we provide a detailed elaboration of this process.

First, Table 1a presents the impact of each fact at the initial stage. The fact with the highest impact is Taint(a), which reveals two insights. (1) Compared with Taint(a), both Taint(t) and Taint(i) are derived through fewer imprecise inference steps. Therefore, they are more likely to hold and thus less likely to be pruned. In fact, they are indeed true facts, and querying them would only waste one interaction with the code agent. (2) Pruning Taint(x1), Taint(x2), Taint(x3), Taint(b), Taint(y1), and Taint(y2) would each affect no more than two alarms, whereas Taint(a) can affect three alarms. Therefore, it is more valuable to let the code agent evaluate Taint(a). Through semantic-level computation, BayPruner predicts that Taint(a) is the most worthwhile fact to be evaluated by the

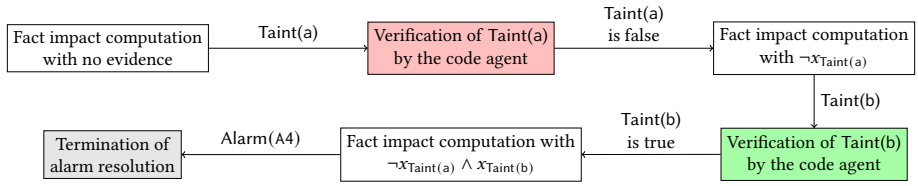


Fig. 4. The full workflow of BayPruner for alarm resolution based on the analysis in Figure 2 applied to the code in Figure 1.

code agent and submits it for verification. This fact is incorrect due to context insensitivity, and the code agent correctly identifies this error. Therefore, we prune it together with all facts derived from it. Since this fact is the common root cause of three alarms, Alarm(A1), Alarm(A2), and Alarm(A3), pruning it removes these alarms and their related facts, and the updated impact ranking is shown in Table 1b.

At this stage, the fact with the highest impact becomes Taint(b), because compared with Taint(y1) and Taint(y2), it can directly affect two alarms. BayPruner then submits it to the code agent for evaluation. This fact is correctly derived, and the code agent confirms its correctness. Since it is a correct fact, we do not prune the derivation graph. Instead, BayPruner feeds the correct judgment of the code agent back to the Bayesian network, which recomputes the probability distribution after learning from the feedback, as shown in Table 1c. Because Taint(y1), Taint(y2), Alarm(A4), and Alarm(A5) are connected to Taint(b), their probabilities of being true increase. As a result, the probabilities that Alarm(A4) and Alarm(A5) do not hold decrease. We assume that $\Pr(\neg x_{\text{Alarm(A4)}} \mid \neg x_{\text{Taint(y1)}} \wedge x_{\text{Taint(b)}}) = \Pr(\neg x_{\text{Alarm(A5)}} \mid \neg x_{\text{Taint(y1)}} \wedge x_{\text{Taint(b)}}) = 0.52$, and $\Pr(\neg x_{\text{Alarm(A4)}} \mid \neg x_{\text{Taint(y2)}} \wedge x_{\text{Taint(b)}}) = \Pr(\neg x_{\text{Alarm(A5)}} \mid \neg x_{\text{Taint(y2)}} \wedge x_{\text{Taint(b)}}) = 0.50$. Consequently, the Weight of Taint(y1) and Taint(y2) decreases to 0.173 and 0.167, respectively.

The key insight at this stage is that Taint(y1) and Taint(y2) are both highly likely to be true facts, and pruning either of them alone cannot eliminate any false alarms. Thus, the benefit of asking the code agent to evaluate them becomes very limited. After learning from the feedback that Taint(b) is true, BayPruner adaptively lowers the priority of Taint(y1) and Taint(y2), thereby correctly determining that their impact has become smaller than directly querying the alarms Alarm(A4) and Alarm(A5). Therefore, BayPruner terminates the filtering process, effectively avoiding the degradation of filtering performance that could even fall below the original naïve one-alarm-at-a-time approach.

Ultimately, BayPruner filters out three false alarms among five alarms by asking the code agent to evaluate only two highly accurate tasks, while fully preserving the two true alarms, achieving both high accuracy and low token consumption.

Next, we present several preliminaries, followed by the formal definition of the fact impact prioritization problem and our Bayesian program analysis based solution, and finally give the formal description of BayPruner based on this solution.

3 Preliminaries

We first introduce the syntax and semantics of Datalog programs, and then describe how they correspond to Datalog-based program analysis.

Figure 5 summarizes the auxiliary notations used in Datalog. Let $\mathbf{V}$ be the set of variables, $\mathbf{D}$ be the set of constants, $\mathbf{R}$ be the set of relation symbols, $\mathbf{L}$ be the set of literals, $\mathbf{T}$ be the set of tuples, and $\mathbf{C}$ be the set of clauses as defined in Figure 5. A literal $p \in \mathbf{L}$ is an n-ary atom $r(a_1, \ldots, a_n)$ that represents a relational atom whose arguments may contain variables or constants. Here a

$$
\begin{aligned}
(\textit{variables}) \quad & \mathbf{V} = \{v, v_1, v_2, s, \dots\} \\
(\textit{constants}) \quad & \mathbf{D} = \{\mathtt{t}, \mathtt{i}, \mathtt{A1}, \mathtt{A2}, 0, 1, \dots\} \\
(\textit{relations}) \quad & \mathbf{R} = \{\mathsf{Input}, \mathsf{Flow}, \mathsf{Memory}, \mathsf{Taint}, \mathsf{Alarm}, \dots\} \\
(\textit{literals}) \quad & \mathbf{L} = \mathbf{R} \times (\mathbf{D} \cup \mathbf{V})^* = \{\mathsf{Taint}(v), \mathsf{Flow}(v_1, v_2), \dots\} \\
(\textit{tuples}) \quad & \mathbf{T} = \mathbf{R} \times \mathbf{D}^* = \{\mathsf{Taint}(\mathtt{t}), \mathsf{Flow}(\mathtt{t}, \mathtt{i}), \dots\} \\
(\textit{clauses}) \quad & \mathbf{C} = \mathbf{L} \times \mathbf{L}^* = \{[l_0 \text{:-} l_1, \dots, l_n], \dots\} \\
& \quad = \{[\mathsf{Taint}(v_2) \text{ :- } \mathsf{Taint}(v_1), \mathsf{Flow}(v_1, v_2)], \dots\}
\end{aligned}
$$

Fig. 5. Auxiliary definitions and notations of Datalog.

$$
\begin{aligned}
F_R, f_c &\in \mathscr{P}(\mathbf{T}) \to \mathscr{P}(\mathbf{T}) \\
F_R(T) &= T \cup \bigcup_{c \in R} f_c(T) \\
f_c(T) &= f_{[l_0 \text{:-} l_1, \dots, l_n]}(T) \\
&= \{\sigma(l_0) \mid \sigma(l_i) \in T \text{ for } 1 \le i \le n, \sigma \in \Sigma\} \\
[[D, T_0]] &= \mathsf{lfp}(F_R, T_0)
\end{aligned}
$$

Fig. 6. Semantics of Datalog.

relation symbol $r \in \mathbf{R}$ denotes a relation type and each $a_i \in \mathbf{V} \cup \mathbf{D}$. A tuple $t \in \mathbf{T}$ is an n-ary atom $r(d_1, \dots, d_n)$ where $r \in \mathbf{R}$ and each $d_i \in \mathbf{D}$. A clause $c \in \mathbf{C}$ has the form $[l_0 \text{ :- } l_1, \dots, l_n]$, which specifies a derivation rule indicating that the head literal l_0 holds whenever the body literals $l_1, \dots, l_n$ hold. A Datalog program consists of input relations, output relations, and a set of derivation rules as defined below:

Definition 3.1 (Datalog program). A Datalog program is a triple $D = (I, O, R)$ where $I \subseteq \mathbf{L}$ is the set of input relations, $O \subseteq \mathbf{L}$ is the set of output relations, and $R \subseteq \mathbf{C}$ is the set of derivation rules.

The semantics of a Datalog program is defined by the rule application operator shown in Figure 6: the operator F_R performs one round of rule application using the rule set R, deriving new tuples from T. For each clause c, the function $f_c(T)$ derives tuples according to substitutions $\sigma \in \Sigma = \mathbf{V} \to \mathbf{D}$. Given an input tuple set T_0, the result of the program, denoted $[[D, T_0]]$, is obtained by iteratively applying the rule operator F_R until reaching the least fixed point.

Definition 3.2 (Datalog semantics). Let $D = (I, O, R)$ be a Datalog program and T_0 a set of input tuples. The semantics of D is computing its result: $[[D, T_0]] = \mathsf{lfp}(F_R, T_0)$.

Program properties capture conditions about program executions, such as "p aliases q", "`len` depends on user input", or "a null pointer dereference occurs at Line 42". These properties may only hold under certain program inputs. Therefore, we use $\textsc{Hold}(p)$ to denote that there exists at least one input for which the execution of the program satisfies the property p.

Definition 3.3 (program property). Let $\mathbf{P}$ denote the set of program properties. We define $\textsc{Hold} : \mathbf{P} \to \{\mathtt{true}, \mathtt{false}\}$ such that $\textsc{Hold}(p)$ indicates that there exists an input under which the execution of the program satisfies the property $p \in \mathbf{P}$.

In practice, the decision of a program property can be implemented by an automated analysis mechanism, such as a code agent. Given a program property p, the mechanism attempts to determine whether p holds, which is formally defined as below:

Definition 3.4 (property judgment). Let $\mathbf{P}$ denote the set of program properties. Consider a decision procedure that determines whether a property holds. For each $p \in \mathbf{P}$, we model the judgment as a random event $\textsc{Judge}(p) \in \{\texttt{true}, \texttt{false}\}$.

Since the judgment may involve heuristic reasoning or incomplete information, the outcome is modeled as a random event. In this work, the decision procedure is instantiated by a code agent. To mitigate the randomness of a single response, each realization of $\textsc{Judge}(p)$ is obtained via self-consistency: the code agent is queried five times independently, and the majority answer is taken as the final judgment. Moreover, we define an accuracy function $\textsc{Acc} : \mathbf{P} \to [0, 1]$ to represent the probability that the mechanism produces a correct judgment for the property p:

$$\forall p \in \mathbf{P}, \quad \textsc{Acc}(p) = \Pr\left[\textsc{Judge}(p) = \textsc{Hold}(p)\right].$$

A Datalog-based program analysis can be viewed as a Datalog program whose derived tuples represent program properties. The mapping function $\textsc{Property}$ associates each tuple in the evaluation result $[\![D, T_0]\!]$ with the corresponding program property. For convenience, we write $[\![\mathcal{D}, T_0]\!]$ to denote the result $[\![D, T_0]\!]$ of the analysis. For each tuple $t \in [\![D, T_0]\!]$, the value $\textsc{Property}(t) \in \mathbf{P}$ denotes the program property corresponding to t. For any $S \subseteq [\![D, T_0]\!]$, the set $\textsc{Alarm}(S) \subseteq S$ contains the tuples in S that correspond to alarm conditions reported by the analysis.

Definition 3.5 (Datalog-based program analysis). A Datalog-based program analysis is defined as a triple $\mathcal{D} = (D, \textsc{Property}, \textsc{Alarm})$, where D is a Datalog program, $\textsc{Property} : [\![D, T_0]\!] \to \mathbf{P}$ is a mapping function, and $\textsc{Alarm} : \mathcal{P}([\![D, T_0]\!]) \to \mathcal{P}([\![D, T_0]\!])$ is a function that maps a set of tuples to the subset representing alarms.

4 The Fact Impact Prioritization Problem and Our Solution

The impact of a fact reflects **how many false alarms can be eliminated by pruning the downstream consequences affected by that fact after querying it**. We **greedily select the fact with the highest impact** so as to remove more false alarms with as few queries as possible. We first give a formal definition of the fact impact prioritization problem, and then present our Bayesian program analysis based solution.

4.1 The Fact Impact Prioritization Problem

Given a Datalog-based program analysis $\mathcal{D}$ and input tuples T_0, a *rule instance* is a derivation using a rule $c = [l_0 \text{ :- } l_1, \ldots, l_n]$. Formally, a rule instance is $([l_0 \text{ :- } l_1, \ldots, l_n], t_0, \ldots, t_n) \in \mathbf{C} \times \mathbf{T}^*$, where each $t_i \in [\![\mathcal{D}, T_0]\!]$ and $\sigma(l_i) = t_i$ for some substitution function $\sigma \in \Sigma$. All rule instances form the set $\textsc{Instance}(\mathcal{D}, T_0)$.

Definition 4.1 (derivation graph). The derivation graph is $\textsc{Graph}(\mathcal{D}, T_0) = (V, E)$, where the vertex set V includes all tuples and rule instances: $V = [\![\mathcal{D}, T_0]\!] \cup \textsc{Instance}(\mathcal{D}, T_0)$. The edge set E represents the dependencies of rules: for each rule instance $i = (c, t_0, \ldots, t_n)$, there are $n + 1$ edges $(t_1, i), \ldots, (t_n, i)$ and (i, t_0).

The derivation graph represents the reasoning process of a Datalog-based program analysis. Each rule instance corresponds to an application of a Datalog rule, linking the tuples in its body to the derived head tuple. Consequently, the graph captures how tuples are derived from others through rule applications. Each tuple $t \in [\![\mathcal{D}, T_0]\!]$ therefore corresponds to a program property obtained through this semantic over-approximation.

We define the pruned derivation graph to model the effect by removing intermediate tuples. Intuitively, pruning removes the tuples in T_p together with all rule instances that depend on them:

Definition 4.2 (pruned derivation graph). Given a derivation graph $\textsc{Graph}(\mathcal{D}, T_0) = (V, E)$ and a set of tuples to prune $T_\mathrm{p} \subseteq [\![\mathcal{D}, T_0]\!]$, let $R \subseteq V$ be the smallest set satisfying

$$T_\mathrm{p} \subseteq R, \quad (t, i) \in E \wedge t \in R \Rightarrow i \in R, \quad \forall i'\big((i', t) \in E \Rightarrow i' \in R\big) \Rightarrow t \in R.$$

The pruned derivation graph is $\textsc{Prune}(\mathcal{D}, T_0, T_\mathrm{p}) = (V', E')$ where

$$V' = V - R, \qquad E' = \{(u, v) \in E \mid u, v \in V'\}.$$

A rule instance is removed if any tuple in its body is removed, and a tuple is removed only when all rule instances that derive it are removed. The removal therefore propagates along the derivation dependencies until reaching a fixed point.

Intuitively, the impact of a fact t measures the expected reduction in false alarms that can be achieved, and we quantify this reduction through the number of pairs of false facts and false alarms:

Definition 4.3 (The set of erroneous intermediate-fact–alarm pairs). Given a pruned derivation graph $\textsc{Prune}(\mathcal{D}, T_0, T_\mathrm{p}) = (V, E)$, let $F = V \cap [\![\mathcal{D}, T_0]\!]$ be the set of facts and $A = \textsc{Alarm}(F)$ be the set of alarms. The set of erroneous intermediate-fact–alarm pairs is defined as

$$\begin{aligned}\textsc{Pairs}(\mathcal{D}, T_0, T_\mathrm{p}) = \{(f, a) \mid\ & f \in F,\ a \in A,\\ & a \text{ is reachable from } f \text{ in } \textsc{Prune}(\mathcal{D}, T_0, T_\mathrm{p}),\\ & \textsc{Hold}(\textsc{Property}(f)) = \texttt{false},\\ & \textsc{Hold}(\textsc{Property}(a)) = \texttt{false}\}.\end{aligned}$$

For a given false alarm, once all the false facts that can reach it in the derivation graph have been pruned, the false alarm itself is pruned as well. Suppose there are n such false facts, i.e., the false alarm participates in n pairs. Then, when all these n pairs become invalid, this false alarm is completely eliminated. Consequently, if only m ($m < n$) of these pairs have been eliminated, we interpret this as having completed a fraction $\frac{m}{n}$ of the progress toward eliminating this false alarm, and assign a contribution of $\frac{m}{n}$ to the impact. Therefore, we formalize this idea as:

Definition 4.4 (The weight of erroneous pairs). For each $(f, a) \in \textsc{Pairs}(\mathcal{D}, T_0, T_\mathrm{p})$, we define its weight by

$$w_{\mathcal{D}, T_0, T_\mathrm{p}}(f, a) = \big|\{f' \mid f' \in F,\ (f', a) \in \textsc{Pairs}(\mathcal{D}, T_0, T_\mathrm{p})\}\big|^{-1}.$$

Finally, after a decision procedure (e.g., a code agent) has judged the program property associated with t to be `false`, summing the weights over all erroneous pairs yields the impact of t. To formalize the expectation, we define an auxiliary random variable $\textsc{Probe}(\mathcal{D}, T_0, T_\mathrm{p}, t)$, which represents a single sampling of the random event $\textsc{Judge}(\textsc{Property}(t))$. If $\textsc{Judge}(\textsc{Property}(t)) = \texttt{false}$, then the value of this random variable is the decrease in the sum of pair weights after pruning fact t:

$$\sum_{(f,a) \in \textsc{Pairs}(\mathcal{D}, T_0, T_\mathrm{p}) - \textsc{Pairs}(\mathcal{D}, T_0, T_\mathrm{p} \cup \{t\})} w_{\mathcal{D}, T_0, T_\mathrm{p}}(f, a).$$

If $\textsc{Judge}(\textsc{Property}(t)) = \texttt{true}$, then the value of the random variable is 0. We define the impact of a fact t as the expectation of this random variable:

Definition 4.5 (fact impact).

$$\textsc{Impact}(\mathcal{D}, T_0, T_\mathrm{p}, t) = \mathbb{E}\big(\textsc{Probe}(\mathcal{D}, T_0, T_\mathrm{p}, t)\big).$$

By always selecting the fact with the highest impact for the code agent to judge, we approximately maximize the reduction in the number of false alarms per query, and thus reduce the total number of queries and the corresponding token consumption.

Definition 4.6 (fact impact prioritization problem). Given a pruned derivation graph $\textsc{Prune}(\mathcal{D}, T_0, T_\text{p}) = (V, E)$, let $F = V \cap [\![\mathcal{D}, T_0]\!]$ be the set of facts. The fact impact prioritization problem is to predict the fact with the maximum impact:

$$\arg\max_{f \in F} \textsc{Impact}(\mathcal{D}, T_0, T_\text{p}, f).$$

4.2 Bayesian Program Analysis for Fact Impact Prioritization Problem

We address the fact impact prioritization problem using Bayesian program analysis [21]. Bayesian program analysis constructs a systematic probability distribution over facts, indicating whether they hold. Based on this distribution, we can directly compute the impact of each fact and thus identify the fact with the largest impact. We first formulate Bayesian program analysis and then explain in detail how to compute the impact based on the probability that each fact holds.

Bayesian program analysis attaches probabilities to rules in the derivation graph to represent their imprecision, thereby converting the derivation graph into a Bayesian network for prediction. Cycles may appear in the derivation graph. Following prior work [21], we remove cycles and treat the graph as a directed acyclic graph (DAG), which simplifies subsequent probabilistic reasoning.

Definition 4.7 (Bayesian network for predicting fact impact). Given a derivation graph $\textsc{Graph}(\mathcal{D}, T_0)$, its Bayesian network is defined as $\textsc{Bayesian}(\mathcal{D}, T_0) = (X, Y)$, where $X = \{x_v \mid v \in V\}$ denotes the set of Bernoulli variables: For each $t \in [\![\mathcal{D}, T_0]\!]$, there exists a corresponding variable $x_t \in X$, and for each rule instance $i = (c, t_0, \ldots, t_n) \in \textsc{Instance}(\mathcal{D}, T_0)$, a variable $x_i \in X$ also exists.

Let $\textsc{Prob}(i) : \mathbf{C} \times \mathbf{T}^* \to [0, 1]$ be the prior probability that a rule instance i is correct. We define the conditional probabilities for the Bernoulli variables as follows:

Definition 4.8 (The conditional probabilities of X). The conditional probabilities are defined separately for the two kinds of variables. (1) *For each rule-instance variable x_i*, where $i = (c, t_0, \ldots, t_n)$ is a rule instance with body tuples $t_1, \ldots, t_n$:

$$\Pr(x_i \mid x_{t_1} \wedge x_{t_2} \wedge \cdots \wedge x_{t_n}) = \textsc{Prob}(i) \qquad \Pr(x_i \mid \neg x_{t_1} \vee \neg x_{t_2} \vee \cdots \vee \neg x_{t_n}) = 0$$

(2) *For each tuple variable x_t*, where t is a tuple derived by rule instances $i_1, \ldots, i_m$:

$$\Pr(x_t \mid x_{i_1} \vee x_{i_2} \vee \cdots \vee x_{i_m}) = 1 \qquad \Pr(x_t \mid \neg x_{i_1} \wedge \neg x_{i_2} \wedge \cdots \wedge \neg x_{i_m}) = 0$$

Then for a given evidence set $\mathcal{E} \subseteq \{\, x, \neg x \mid x \in X \,\}$ and for any tuple $t \in [\![\mathcal{D}, T_0]\!]$, $\Pr\left(x_t \mid \bigwedge_{e \in \mathcal{E}} e\right)$ is the conditional probability of x_t on evidence $\mathcal{E}$.

Note that we compute $\Pr\left(x_t \mid \bigwedge_{e \in \mathcal{E}} e\right)$ using a probabilistic inference algorithm on the Bayesian network [15], and we denote this result by $\textsc{Inference}(t, \mathcal{E})$. Intuitively, x_t indicates whether $\textsc{Hold}(\textsc{Property}(t)) = \texttt{true}$. Similarly, x_i indicates whether $\textsc{Hold}(\textsc{Property}(t_0)) = \texttt{true}$, conditioned on $\textsc{Hold}(\textsc{Property}(t_1)), \ldots, \textsc{Hold}(\textsc{Property}(t_n))$ all being $\texttt{true}$. The prior probabilities $\textsc{Prob}$ are theoretically well-defined but are intractable to obtain exactly in real-world settings. In practice, we estimate them via expert-provided annotations [21] or data-driven methods trained on labeled programs [10]. In this way, the theoretical framework remains coherent and principled, while a practical instantiation is achieved by approximating the prior.

Given the pruned derivation graph $\textsc{Prune}(\mathcal{D}, T_0, T_\text{p}) = (V, E)$, for each fact t and each alarm a, we define $\textsc{Reduce}(\mathcal{D}, T_0, T_\text{p}, t, a)$ as follows: it represents the reachable pairs (f, a) are eliminated when t is pruned, where each pair is counted irrespective of whether f and a hold. Formally,

$$\begin{aligned}\textsc{Reduce}(\mathcal{D}, T_0, T_\text{p}, t, a) = \{(f, a) \mid\ & f \in V \cap [\![\mathcal{D}, T_0]\!], \\ & f \text{ is reachable to } a \text{ in } \textsc{Prune}(\mathcal{D}, T_0, T_\text{p}), \\ & f \text{ is not reachable to } a \text{ in } \textsc{Prune}(\mathcal{D}, T_0, T_\text{p} \cup \{t\})\}.\end{aligned}$$

Since we do not consider the truth values of facts, $\textsc{Reduce}(\mathcal{D}, T_0, T_p, t, a)$ can be computed directly by counting on the graph.

Next, we show how to compute the fact impact using the predictions of the Bayesian network given the evidence set $\mathcal{E}$. We denote this result by $\textsc{Impact}_{\mathcal{E}}(\mathcal{D}, T_0, T_p, t)$. We have:

$$\begin{aligned}\textsc{Impact}_{\mathcal{E}}(\mathcal{D}, T_0, T_p, t) = \textsc{Acc}(\textsc{Property}(t)) \cdot \sum_{a \in \textsc{Alarm}(V \cap [\![\mathcal{D}, T_0]\!])} (1 - \textsc{Inference}(a, E)) \cdot \\ (1 - \textsc{Inference}(t, E \cup \{\neg x_a\})) \cdot \sum_{(f,a) \in \textsc{Reduce}(\mathcal{D}, T_0, T_p, t, a)} w_{\mathcal{D}, T_0, T_p}(f, a).\end{aligned} \tag{1}$$

We provide a detailed calculation of Equation 1 in the appendix. Thus, given a pruned derivation graph $\textsc{Prune}(\mathcal{D}, T_0, T_p) = (V, E)$, let $F = V \cap [\![\mathcal{D}, T_0]\!]$ be the set of facts, we can compute the fact with the largest impact:

$$\arg\max_{f \in F} \textsc{Impact}_{\mathcal{E}}(\mathcal{D}, T_0, T_p, f).$$

5 The BayPruner Framework

We first present the problem we aim to solve, and then formalize BayPruner.

Let $\mathcal{D}$ be a Datalog-based program analysis and T_0 the input tuples. We aim to select a subset of facts $F \subseteq [\![\mathcal{D}, T_0]\!]$ to query a code agent. For each $t \in F$, we perform one sampling of the random event $\textsc{Judge}(\textsc{Property}(t))$, which is realized as the majority vote over five independent code-agent judgments (Definition 3.4), and let

$$T_p = \{\, t \in F \mid \textsc{Judge}(\textsc{Property}(t)) = \texttt{false} \,\}.$$

Let $\textsc{Prune}(\mathcal{D}, T_0, T_p) = (V, E)$ be the pruned derivation graph, and define the remaining alarm set as

$$A = \textsc{Alarm}\big(V \cap [\![\mathcal{D}, T_0]\!]\big).$$

Our goal is to choose $F \subseteq [\![\mathcal{D}, T_0]\!]$ to (1) maximize the number of removed false alarms (i.e., $\textsc{Hold}(\textsc{Property}(a)) = \texttt{false}$) and the number of preserved true alarms (i.e., $\textsc{Hold}(\textsc{Property}(a)) = \texttt{true}$), while (2) minimizing the number of code-agent queries, i.e., minimizing $|F|$. BayPruner uses Bayesian program analysis to compute the impact of each fact, and **greedily queries the fact with the largest impact that can be judged by code agents with high accuracy**. The procedure continues until the highest-impact fact is itself an alarm; to avoid degenerating into the one-alarm-at-a-time method, BayPruner terminates alarm resolution at this point.

Algorithm 1 illustrates the workflow of BayPruner. First, BayPruner maintains the set of tuples that have already been pruned, denoted by T_p, and the current set of feedback evidence $\mathcal{E}$ (both initialized to be empty) (Line 2). Then BayPruner enters an iterative loop. In each iteration, it first computes the current pruned derivation graph (Line 4) and then computes the remaining set of alarms A (Line 5). If A is empty (Line 6), this means that all alarms have been pruned, i.e., all alarms are judged by BayPruner to be false alarms, so BayPruner directly returns the empty set and terminates (Line 7).

Otherwise, for the remaining facts, BayPruner uses the Bayesian program analysis, based on the current evidence $\mathcal{E}$, to compute the impact of each fact. We pre-estimated $\textsc{Acc}(\textsc{Property}(t))$, namely the accuracy with which the code agent can judge it to be correct for each fact t. It is obtained via offline learning from the training set as discussed in Section 6.1. Among all facts whose code agents have a pre-estimated accuracy no lower than $\textsc{MinAcc} \in [0, 1]$ (a parameter deciding the precision), BayPruner selects the fact t with the largest impact as the query target for this round (Line 9). If the impact of this fact does not exceed the maximum impact among the remaining alarms (Line 10), then BayPruner considers that further queries would degenerate into the original

Algorithm 1 The BayPruner framework.

Input: The Datalog-based program analysis $\mathcal{D}$,the input tuples T_0, and the Bayesian network B.
Output: The alarm set A after resolution.

1: **procedure** BayPruner($\mathcal{D}, T_0, B$)
2: $T_\text{p} \leftarrow \emptyset, \mathcal{E} \leftarrow \emptyset$
3: **while** True **do**
4: Let (V, E) = Prune($\mathcal{D}, T_0, T_\text{P}$)
5: A = Alarm$\big(V \cap [\![\mathcal{D}, T_0]\!]\big)$
6: **if** $A = \emptyset$ **then**
7: **return** $\emptyset$
8: $T \leftarrow V \cap [\![\mathcal{D}, T_0]\!]$
9: $t \leftarrow \arg\max_{t' \in T, \text{Acc}(\text{Property}(t')) \geq \text{MinAcc}} \text{Impact}_\mathcal{E}(\mathcal{D}, T_0, T_\text{p}, t')$
10: **if** $\text{Impact}_\mathcal{E}(\mathcal{D}, T_0, T_\text{p}, t) \leq \max_{a \in A} \text{Impact}_\mathcal{E}(\mathcal{D}, T_0, T_\text{p}, a)$ **then**
11: **return** A
12: $e \leftarrow$ A single sample of the random event Judge(Property(t)), i.e., the majority vote over five independent code-agent judgments
13: **if** e = `false` **then**
14: $T_\text{p} \leftarrow T_\text{p} \cup \{t\}$
15: $\mathcal{E} \leftarrow \mathcal{E} \cup \{\neg x_t\}$
16: **else**
17: $\mathcal{E} \leftarrow \mathcal{E} \cup \{x_t\}$

one-alarm-at-a-time method, and thus terminates the querying process and returns the current remaining alarm set A (Line 11).

Otherwise, BayPruner continues by asking the code agents to judge whether this fact holds (Line 12). If they judge that it does not hold (Line 13), BayPruner prunes the corresponding tuple from the derivation graph (Line 14). Finally, BayPruner collects this feedback and adds it to the evidence set $\mathcal{E}$ (Line 15 and Line 17), and then starts the next iteration.

Note that pruning is performed only on the derivation graph, which determines the remaining alarms and the impact computation (Section 4); the structure of the Bayesian network is never modified. Instead, each judgment of the code agent enters the network as evidence: when a fact t is judged false, BayPruner conditions the network on $\neg x_t$ as negative evidence, and the posterior beliefs of all dependent facts and alarms are updated via belief propagation in the next round of impact computation. In particular, conditioning on $\neg x_t$ drives the posteriors of all tuples that are derivable only through t to zero, keeping the probabilistic model consistent with the pruned derivation graph.

Theorem 5.1. *For every alarm deleted by BayPruner, the probability that it is a true alarm mistakenly deleted does not exceed* $1 - \textit{MinAcc}$. *Formally, for the returned result A, for any* $a \in \textit{Alarm}([\![\mathcal{D}, T_0]\!] \setminus A)$, *we have*

$$\Pr\big(\textit{Hold}(\textit{Property}(a)) = \texttt{true}\big) \leq 1 - \textit{MinAcc}.$$

Proof. If a is a true alarm that gets deleted, then there exist n facts $f_1, f_2, \ldots, f_n$ that can reach a in the derivation graph, and the code agent must have made an error on each of these facts when judging them. Since

$$\text{Acc}(\text{Property}(t)) \geq \text{MinAcc}$$

for all queried facts t, the probability of this event is at most

$$(1 - \text{MinAcc})^n \leq 1 - \text{MinAcc}.$$

□

BayPruner achieves high-accuracy, low-token-cost alarm resolution by querying only high-accuracy facts to optimize objective (1), and by systematically solving the fact impact prioritization problem via Bayesian program analysis to optimize objective (2).

6 Experimental Evaluation

Our evaluation aims to answer the following questions:

RQ1. Can BayPruner achieve an effective combination of high accuracy and low token consumption in alarm resolution?
RQ2. Does the use of different LLMs affect the performance of BayPruner?
RQ3. Does the inherent randomness of LLMs affect the performance of BayPruner?
RQ4. BayPruner leverages Bayesian program analysis to address the fact impact prioritization problem. Does this module contribute to the effectiveness of BayPruner?

We first describe our experimental setup in Section 6.1. Then, we answer the four research questions in Section 6.2 to Section 6.5, respectively.

6.1 Experimental Setup

We conducted all experiments on Linux machines with 2.6 GHz processors and 256 GB RAM. We use libDAI [14] to perform probabilistic inference on the Bayesian network. We use Codex 0.66.0 [19] (GPT-5.1-codex-max medium [20]) as the code agent. We set the temperature to the default value of 1.0. Figure 7 shows the prompt template we use. To mitigate the impact of LLM randomness, we repeat the judgment for each fact five times and adopt the majority vote as the final decision.

Instance analyses. We report summary statistics for the two instance analyses in Table 2. We conducted experiments on two analyses of different types and in different programming languages to fully demonstrate the generality of our approach: (1) The first analysis is a datarace analysis for Java programs built on top of the Chord framework [16]. It identifies all pairs of statements that can concurrently access the same heap object, where at least one of the accesses may perform a write. This analysis integrates flow- and context-sensitive thread-escape, may-happen-in-parallel, and lockset analyses, and it employs a 3-context-and-object-sensitive but flow-insensitive pointer analysis [13] to construct the call graph and aliasing information. To better approximate realistic scenarios in alarm resolution, we enabled the unsound option to reduce the number of reported alarms as much as possible. (2) The second analysis is a taint analysis for C programs based on the Sparrow framework [18]. It checks whether attacker-controlled format strings and integer values that may overflow are passed as arguments to `printf`-like and `malloc`-like functions, respectively. This analysis leverages the sparse analysis framework [17] to construct def–use relations among statements in order to enable taint propagation.

Benchmarks. Table 3 lists the benchmarks used in our experiments. These benchmarks are collected from prior work [5, 7, 8, 21], the DaCapo suite [3], and CVE reports. They are widely used in static-analysis research, and tool performance on them is therefore representative of its behavior in real-world engineering settings. All reported alarms were manually inspected to identify the true positives.

Baselines. We compare against the following two baselines: (1) Base-J, which submits all alarms to the code agent for judgment in sequence and retains only those that the agent deems correct; and (2) Base-S, which directly uses the static-analysis results without any filtering by the code agent. We compare with Base-S to show that our method does not lead to negative optimization relative to conventional workflows with LLMs.

System: You are a static analysis expert with exceptional code comprehension abilities. You are capable of rigorously reasoning about program semantics to determine whether a given program property can hold during any feasible execution of the code.

Prompt Content: I will provide a program property, such as "A may point to B", "The value of C may be 1", etc. Based on the contents of the codebase, please determine whether there exists an input that satisfies the property I provide. At the end, you must give your answer: "exists" or "does not exist". The provided program property is:

<PROGRAM PROPERTY>

Fig. 7. Prompt template for instructing the code agent to determine whether a program property holds.

Table 2. Statistics of the instance analyses.

Analysis	# Input relations	# Output relations	# Derivation rules
Datarace analysis	58	44	102
Taint analysis	6	9	14

Table 3. Benchmark characteristics. In the **# True alarms** column, x/y means that among the alarms reported by the analysis, x correspond to real bugs, and the program contains y real bugs of this type in total.

Program		Description	KLOC	# Alarms	# True alarms
Datarace analysis	ftp	Apache FTP server	305	211	36/75
	jspider	Web spider engine	298	52	5/9
	hedc	Web crawler from ETH	292	55	6/12
	weblech	Website download/mirror tool	322	7	4/6
	avrora	AVR microcontroller simulator	318	9	4/29
	luindex	Document indexing tool	299	494	2/2
	sunflow	Photo-realistic image rendering system	529	171	94/171
	xalan	XML to HTML transforming tool	495	80	52/75
Taint analysis	jhead	Exif Jpeg header manipulation tool	5	19	2/2
	shntool	Command-line WAVE data viewer/modifier	13	23	6/6
	autotrace	Bitmap to vector graphics converter	18	77	18/18
	sam2p	Raster (bitmap) image converter	22	20	12/12
	sdop	Simple docbook processor	23	150	65/65
	latex2rtf	Translator from LaTeX to RTF	27	13	2/2
	urjtag	Universal JTAG library, server and tools	46	35	6/6
	optipng	PNG optimizer that recompresses image files	61	67	1/1
	a2ps	Text to PostScript formatter	64	27	6/6

Hyperparameter configuration. BayPruner has three groups of hyperparameters: the rule probabilities Prob, the per-relation judging accuracies Acc, and the pruning threshold MinAcc. All of them are configured in a data-driven manner. (1) For Prob, following prior work on Bayesian program analysis [10, 12, 21], we estimate the rule probabilities using the expectation–maximization (EM) algorithm with leave-one-out cross-validation across programs: when running BayPruner on a target program, the probabilities are learned from all other programs of the same analysis. The exact values are not critical, as what matters is separating high-confidence rules from low-confidence ones: perturbing every estimated probability by ± 0.05 does not change the final pruning results on any benchmark. Moreover, as the code agent's judgments accumulate in the evidence set $\mathcal{E}$, the posterior distribution is increasingly dominated by this high-confidence evidence, further diminishing the sensitivity to the exact prior values. (2) For Acc, we construct a labeled set by randomly sampling facts of each relation type from each program's analysis output and manually annotating whether each sampled fact holds. The code agent is then queried on every sampled fact (five samples with majority voting), and its per-relation accuracy is measured against the

Table 4. The estimated accuracy (Acc, in %) of the code agent on each relation type, represented by its corresponding program property and sorted in descending order. Relation types from the taint analysis are marked with †; the rest are from the datarace analysis.

Program property	Acc	Program property	Acc
A statement is a method invocation.	100.0	A static field may hold an object created under a call context.	91.8
An invocation calls `Thread.start()`.	100.0	A memory access may happen in parallel with another thread.	91.4
A field is static.	100.0	A variable may point to an object created under another call context.	90.8
Same-thread pairs are excluded from reporting.	100.0	A thread is created at a statement.	89.2
A heap access occurs at a statement.	99.8	A created thread starts with a root method.	88.6
A memory access touches a field or an array slot.	99.7	A condition evaluates to true toward a statement.†	87.9
A method starts at a statement.	99.6	A condition evaluates to false toward a statement.†	87.4
A method ends at a statement.	99.6	A call may enter a method running under another call context.	86.7
A method argument is stored in a local variable.	99.5	Control returns from a call to the following statement.	85.2
A memory access touches a static field.	99.4	A thread-start call begins a method in a new thread.	82.3
The true branch transfers control between two statements.†	98.3	A statement starts a new thread under a call context.	81.7
The false branch transfers control between two statements.†	98.1	A new thread begins at a statement under a call context.	80.4
Control flows from one statement to the next.	97.8	Reading a field may yield an object created under another call context.	78.9
A def-use edge connects two statements.†	96.9	A statement may execute in parallel with another thread.	68.9
A memory access reads or writes the object in a local variable.	96.4	Another thread may run in parallel when a call returns.	66.4
A call statement enters a method under a call context.	94.6	Two parallel memory accesses conflict without a common lock.	52.7
A created thread starts executing at a statement.	94.1	Two parallel memory accesses touch shared objects.	50.2
A call enters the entry statement of its callee under a call context.	93.8	Two memory accesses from parallel threads may race.	47.5
A data flow between two statements may lead to a vulnerability.†	93.5	A data race exists between two accesses under call contexts.	45.9
A memory access is visible to other threads.	93.2	A data race exists between two memory accesses.	44.6
An object escapes the thread that creates it.	92.6	A tainted value may reach a sensitive sink.†	43.8
A memory access may reach an object created under a call context.	92.4	A taint alarm is reported at a statement.†	43.5
Another thread may run in parallel when a statement executes.	92.1		

manual labels, also under leave-one-out cross-validation: the accuracy used for a target program is estimated on the labeled samples of all other programs. Table 4 reports the measured accuracies of the 45 relation types that appear in the derivation graphs of our benchmarks. The code agent is nearly perfect on purely syntactic properties, highly reliable on local control- or data-flow facts, single-step semantic facts, and a number of multi-component facts, while substantially less reliable on the remaining context-sensitive facts, and least reliable on end alarms such as race pairs and taint alarms. (3) MinAcc expresses the user's confidence requirement before pruning: by the theorem in Section 5, every pruned alarm is a mistakenly deleted true alarm with probability at most $1 - \textsc{MinAcc}$. It can be selected by inspecting the estimated per-relation accuracies on the training programs. We set $\textsc{MinAcc} = 0.9$, which cleanly separates the relation types that the code agent judges reliably from those it does not: only tuples with $\textsc{Acc} \geq 0.9$ are submitted to the code agent.

Metrics. We use the following three metrics to evaluate the accuracy of different alarm resolution methods:

$$\text{Precision} = \frac{\left|\{\, a \mid a \in A,\ \textsc{Hold}(\textsc{Property}(a)) = \texttt{true} \,\}\right|}{|A|},$$

Table 5. Accuracy, number of queries, and token consumption of BayPruner and the baselines on the benchmarks. Columns **BP**, **J**, and **S** denote BayPruner, Base-J (asking the LLM to judge each alarm), and Base-S (running static analysis without LLM filtering), respectively. The **Average** row reports the average multiplicative improvement of BayPruner over the corresponding baseline for accuracy, and the average relative reduction for the number of queries and token consumption.

	Program	Precision			Recall			F1-score			# Queries		# Tokens	
		BP	J	S	BP	J	S	BP	J	S	BP	J	BP	J
Datarace analysis	ftp	**0.30**	0.20	0.17	**0.48**	0.47	**0.48**	**0.37**	0.28	0.25	**4**	211	**422,882**	19,330,767
	jspider	0.12	**0.18**	0.10	**0.56**	0.22	**0.56**	0.19	**0.20**	0.16	**4**	52	**1,845,876**	15,502,655
	hedc	0.17	**0.83**	0.11	**0.50**	0.42	**0.50**	0.26	**0.56**	0.18	**4**	55	**745,948**	5,304,422
	weblech	**0.80**	0.57	0.50	**0.67**	**0.67**	**0.67**	**0.73**	0.61	0.57	**2**	7	**498,669**	747,461
	avrora	**0.57**	0.25	0.44	**0.14**	0.03	**0.14**	**0.22**	0.06	0.21	**4**	9	**704,954**	4,247,729
	luindex	**0.18**	0.10	0.00	**1.00**	**1.00**	**1.00**	**0.31**	0.18	0.01	**16**	494	**8,886,298**	145,001,921
	sunflow	0.32	**0.62**	0.20	**0.58**	0.09	**0.58**	**0.41**	0.15	0.29	**3**	506	**443,092**	125,584,090
	xalan	**1.00**	0.00	0.65	**0.69**	0.00	**0.69**	**0.82**	0.00	0.67	**9**	80	**3,357,510**	5,427,534
	Average	-	1.20 ×	7.01 ×	-	2.48 ×	1.00 ×	-	1.71 ×	5.94 ×	-	86.88%↓	-	77.52%↓
Taint analysis	jhead	**0.40**	0.08	0.10	**1.00**	0.50	**1.00**	**0.57**	0.14	0.19	**5**	19	**536,771**	3,261,592
	shntool	**0.32**	0.30	0.26	**1.00**	**1.00**	**1.00**	**0.48**	0.46	0.41	**4**	23	**721,924**	3,822,743
	autotrace	**0.44**	0.25	0.23	**1.00**	**1.00**	**1.00**	**0.61**	0.40	0.38	**5**	77	**1,328,857**	14,200,275
	sam2p	**0.67**	0.60	0.60	**1.00**	**1.00**	**1.00**	**0.80**	0.75	0.75	**7**	20	**248,241**	845,767
	sdop	**0.51**	0.43	0.43	0.83	**1.00**	**1.00**	**0.63**	0.60	0.60	**50**	150	**5,871,213**	21,499,008
	latex2rtf	**0.40**	0.15	0.15	**1.00**	**1.00**	**1.00**	**0.57**	0.27	0.27	**5**	13	**568,245**	1,406,521
	urjtag	**0.25**	0.23	0.17	**1.00**	**1.00**	**1.00**	**0.40**	0.38	0.29	**24**	35	**2,440,858**	7,681,471
	optipng	**0.25**	0.02	0.01	**1.00**	**1.00**	**1.00**	**0.40**	0.03	0.03	**5**	67	**431,898**	28,503,910
	a2ps	**0.27**	0.20	0.12	**1.00**	**1.00**	**1.00**	**0.43**	0.33	0.21	**12**	26	**2,533,906**	5,897,102
	Average	-	3.30×	3.59×	-	1.09×	0.98×	-	2.77×	3.03×		68.98%↓	-	75.78%↓

$$\text{Recall} = \frac{\left|\{\, a \mid a \in A,\ \textsc{Hold}(\textsc{Property}(a)) = \texttt{true} \,\}\right|}{\left|\{\, a \mid a \in \textsc{Alarm}([[\mathcal{D}, T_0]]),\ \textsc{Hold}(\textsc{Property}(a)) = \texttt{true} \,\}\right|},$$

$$\text{F1-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}.$$

We measure the cost using the number of queries and the total token consumption.

6.2 Effectiveness

Table 5 reports the accuracy, number of queries, and token consumption of BayPruner and the two baselines on each program. Overall, compared with Base-J, BayPruner improves the F1-score by 1.71× and 2.77× on the data-race analysis and the taint analysis, respectively, while reducing the number of queries to 86.88% and 68.98% fewer than Base-J, thereby cutting token consumption by 77.52% and 75.78%. These results show that BayPruner successfully addresses the key limitation of the one-alarm-at-a-time method, enabling LLM-based alarm resolution with both high accuracy and low token cost.

Compared with Base-S, BayPruner improves the F1-score by 5.94× and 3.03×, respectively. This substantial gain indicates that BayPruner is an effective and robust practical alternative to conventional static-analysis-only workflows. Furthermore, except for sdop in the taint analysis, BayPruner does not remove any true positives on the other programs, achieving the same recall as Base-S. In fact, across all 17 programs, BayPruner issued 163 fact queries in total, of which only 10 were judged incorrectly—all on sdop—yielding an overall accuracy of 93.9%, consistent with the estimated per-relation accuracies and the threshold MinAcc = 0.9. This suggests that, by restricting query types based on the code agents' estimated accuracy, BayPruner effectively avoids the negative impact of inaccurate judgments on true alarms. In contrast, because the code agent cannot reliably judge alarms, Base-J mistakenly discards a large number of true positives, leading to a substantial drop in recall.

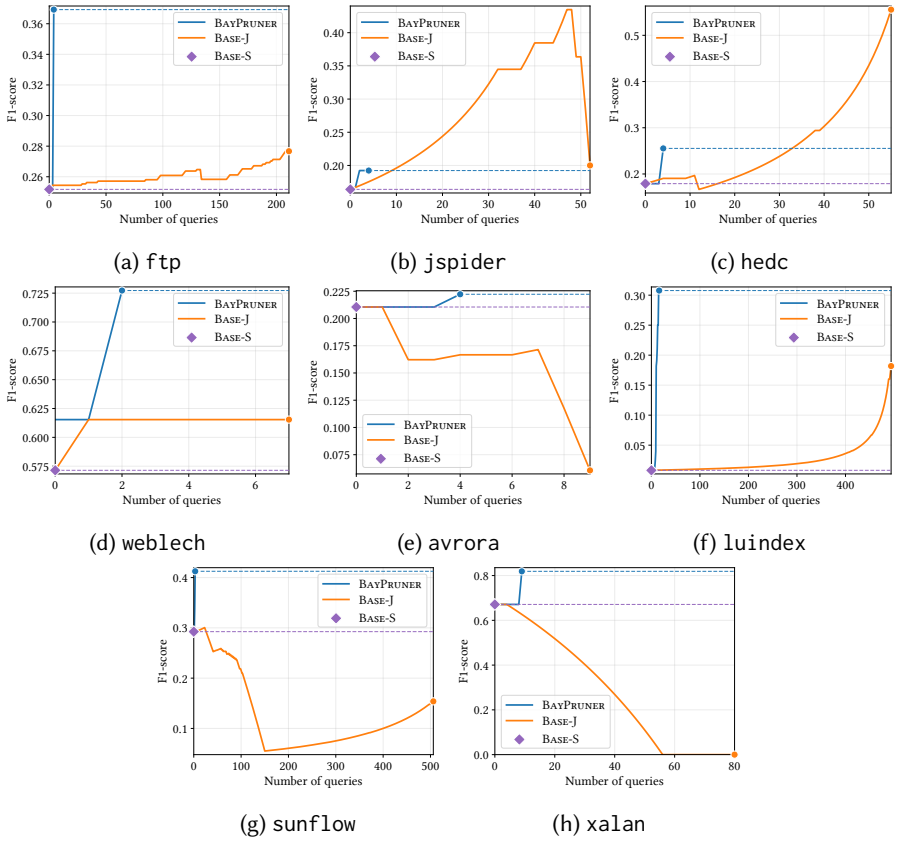


Fig. 8. F1-score of BayPruner and baselines for the datarace analysis as they progressively filter alarms with an increasing number of queries. The dashed lines indicate that the corresponding method stops further filtering after that point.

There are two outliers where BayPruner attains a lower F1-score than Base-J on `jspider` and hedc. This is because these programs have relatively small core code bases (only 6.7 and 6 KLOC excluding the JDK), and the alarms involve short reasoning chains; consequently, the code agents can directly judge alarms with relatively high accuracy. However, this special case mainly applies to small programs; for large and complex real-world software, Base-J does not perform well in general.

Furthermore, Figure 8 and Figure 9 visualize how the F1-score of different methods evolves with an increasing number of queries for the two analyses. Initially, when no queries are issued, all three methods yield identical filtering results. We observe that, in most cases, BayPruner can substantially improve the F1-score with only a small number of queries, and then terminates further filtering. In contrast, Base-J exhibits large fluctuations; on many programs it even shows clear negative optimization, with an F1-score significantly lower than that of the unfiltered baseline Base-S. The low accuracy of the code agent on alarms not only wastes a large number of tokens, but also leads to severe instability and adverse effects. By contrast, BayPruner effectively avoids these two critical issues.

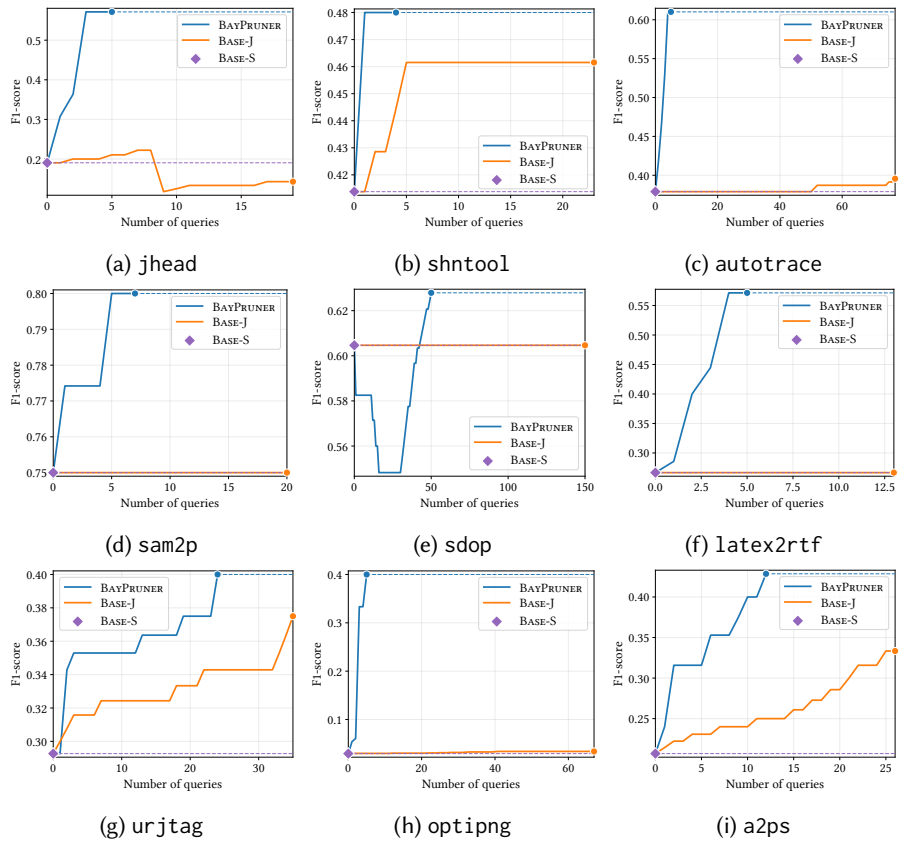


Fig. 9. F1-score of BayPruner and baselines for the taint analysis as they progressively filter alarms with an increasing number of queries. The dashed lines indicate that the corresponding method stops further filtering after that point.

In conclusion, compared with the baselines, BayPruner achieves alarm resolution with both high accuracy and low token consumption.

6.3 Impact of Different LLMs

We compared the performance changes of BayPruner after replacing the code agent with Claude Code 2.1.68 [1] (Claude-Sonnet-4.6 [2]). We set the temperature to the default value of 1.0. The results are shown in Figure 10. The results indicate that the average F1-score decreased by 0.1%, and the average number of queries decreased by 2.4%, showing almost no impact. Moreover, for most programs, the results did not change at all. The changes observed on a small number of programs are due to differences in how different code agents judge program properties, which is entirely acceptable. Therefore, the type of large language model has no effect on the performance of BayPruner.

6.4 Impact of LLM Randomness

To examine whether the inherent randomness of LLMs can affect the performance of BayPruner, we report, over five repeated code-agent queries, how often the majority answer is “exists” versus

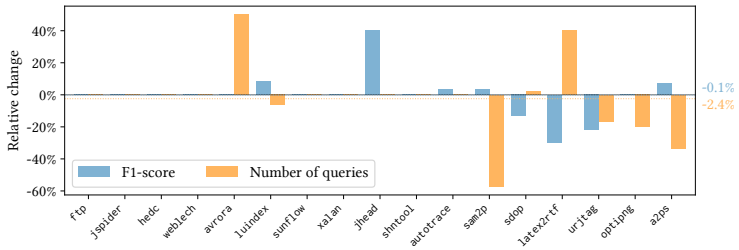
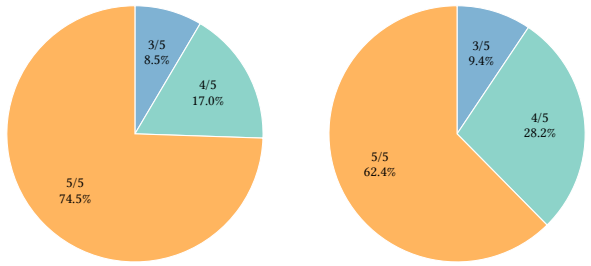


Fig. 10. The relative change rates of F1-score and the number of queries for BayPruner when using Claude Code compared to Codex. The dashed line denotes the average value.



(a) Datarace analysis. (b) Taint analysis.

Fig. 11. Agreement with the majority answer across five repeated runs for BayPruner on two analyses.

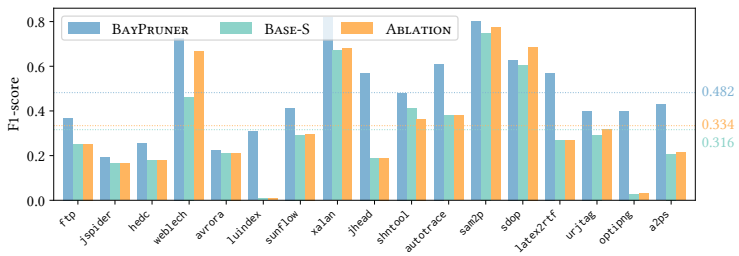


Fig. 12. F1-scores of BayPruner, Base-S, and Ablation across different programs. The dashed lines indicate the average value.

"does not exist"; the results are shown in Figure 11. The results indicate that, in the vast majority of cases, all five invocations return the same answer, and only in a few cases do one or two answers differ. This suggests that, for program properties that the code agent can judge with high accuracy, the uncertainty of the agent is also relatively low, enabling stable judgments in most cases. Overall, the intrinsic randomness of LLMs has negligible impact on BayPruner's performance.

6.5 Ablation Study

We conducted an ablation study to investigate the necessity of selecting high-impact facts. We use Ablation to denote a variant that replaces high-impact fact selection with uniformly random

selection among high-accuracy facts, i.e., replacing Line 9 in Algorithm 1 with

$$t \leftarrow \text{Random select from } \{\, t' \in T \mid \textsc{Acc}(\textsc{Property}(t')) \geq \textsc{MinAcc} \,\}.$$

In addition, we manually constrain Ablation to use the same number of queries as BayPruner. Figure 12 compares this setting with BayPruner and Base-S. As shown, Ablation is almost indistinguishable from Base-S, and its F1-score is substantially lower than that of BayPruner. The reason is that the derivation graph contains far more facts than alarms, and most alarms have multiple derivation paths. If facts are selected at random, (i) the queries are likely to hit facts that do hold, and (ii) it is difficult to prune multiple alarms simultaneously. Consequently, almost no alarms are pruned. Therefore, selecting high-impact facts based on Bayesian program analysis is an effective and necessary component of BayPruner.

7 Discussion

We discuss (1) the deployment of BayPruner beyond Datalog-based analyses and (2) the comparison between our impact formulation and a causal/interventional formulation.

Deployment beyond Datalog-based analyses. BayPruner only requires the derivation graph of an analysis, i.e., the derived facts and the dependencies among them. While Datalog-based analyzers expose this structure natively, Bayesian program analysis has also been successfully deployed on non-Datalog analyzers. For example, a recent work [28] applies Bayesian program analysis to the taint analysis of SVF [24], an industrial-strength non-Datalog static analyzer for LLVM, and runs it on widely fuzzed industrial-grade open-source projects. The engineering effort of such deployment is modest: any static analyzer continuously derives new facts during its execution, so constructing the derivation graph only requires additionally recording the dependency relations among these facts. In the deployment of [28], the derivation graph is built around facts of the form "there is a data flow from x to y," which are readily available inside SVF. The runtime overhead is also low: we ran the publicly available artifact of [28] and measured that generating the derivation graph introduces only about 5% time overhead relative to the full static analysis.

Comparison with a causal/interventional formulation. Our impact metric can be understood from a causal perspective: pruning a fact t corresponds to the intervention $do(x_t = \texttt{false})$ on the model induced by the derivation graph, and the effect of this intervention is exactly what $\textsc{Prune}(\mathcal{D}, T_0, T_p \cup \{t\})$ computes. A direct interventional objective would rank facts by the expected number of false alarms eliminated by the intervention. However, this objective does not yield an effective strategy in our setting, because the derivation graph mixes AND-dependencies (a rule instance requires all its body tuples) with OR-dependencies (a tuple survives if any deriving rule instance survives): removing a single fact typically eliminates no alarm by itself (Section 2), so the direct objective is zero for almost all candidate facts and provides no ranking signal. The principled global problem is instead to select a minimum set of facts whose removal eliminates all false alarms; this problem is NP-complete even given ground-truth labels of all facts and alarms, and is strictly harder without ground truth. Our impact formulation therefore adopts the expected reduction of erroneous fact–alarm reachability pairs as a tractable proxy: it still measures the interventional effect of pruning, while additionally rewarding partial progress toward eliminating each false alarm, capturing cross-alarm dependencies, and admitting efficient greedy ranking.

8 Related Work

Our approach is related to (1) Bayesian program analysis and (2) techniques that leverage LLMs to enhance alarm resolution. We summarize the related work below.

Bayesian program analysis. Bayesian program analysis transforms the derivation process of conventional program analysis into a Bayesian model by attaching probabilities to derivation rules.

The resulting Bayesian model can incorporate various kinds of posterior information and ultimately outputs, for each alarm, a probability distribution of being true. Bingo [21] first introduced Bayesian program analysis and used user feedback as posterior information. Drake [8] proposed learning information from previous code versions, DynaBoost [5] proposed learning from dynamic testing information, and NESA [12] proposed learning from informal information. BayeSmith [10] proposed parameterized learning of the model structure from a syntactic perspective. BinGraph [27] and BayesRefine [23] proposed structured learning of the model structure from the perspective of abstraction selection. These works focus on producing alarm-truth probability distributions that better match reality, but they lack a systematic method for alarm resolution based on such probability distributions. BayPruner fills this gap by systematically leveraging the probability distributions produced by Bayesian program analysis to efficiently prune alarms using high-impact facts. Therefore, the above works are orthogonal to BayPruner: more accurate probability distributions produced by them can further improve the effectiveness of BayPruner. In addition, Bayzzer [28] extends the semantics of Bayesian program analysis to guide fuzzing in selecting suitable targets from a set of alarms; BayPruner can be naturally combined with Bayzzer by performing alarm resolution in advance to further narrow the target selection space.

LLM-enhanced alarm resolution. Recent studies have explored using LLMs to mitigate false positives in static analysis. Existing works such as LLM4SA [25] and LLM4FPM [4] focus on context engineering after the target has been fixed, typically an end alarm. They assume the alarm to validate is already given, and then optimize the code context presented to the LLM so it can judge that specific alarm more accurately. In contrast, BayPruner tackles an earlier decision: selecting what the LLM should validate in the first place. Instead of checking end alarms directly, BayPruner plans LLM queries at the semantic level by selecting critical intermediate derived facts. Therefore, The two directions are complementary. Although some other methods involve semantics, they are customized for specific analyses. ZeroFalse [9] guides LLMs using dataflow traces enriched with CWE-specific knowledge. LLM4PFA [6] uses LLM agents for constraint reasoning and context-aware analysis. It validates path feasibility and removes alarms on infeasible paths. In contrast, BayPruner operates directly on the derivation graph, a universal abstraction of logical reasoning. This design makes it applicable to almost all static analyses and provides significantly better generality.

9 Conclusion

We propose BayPruner, an LLM-based alarm resolution framework guided by Bayesian program analysis. BayPruner expands a code agent's query scope from individual alarms to facts that can be judged with high accuracy, and uses the probability distributions from the Bayesian model to select high-impact facts for the code agent to inspect, thereby achieving alarm resolution with both high accuracy and low token consumption. We evaluate BayPruner on two representative analyses across different languages and analysis types, demonstrating its effectiveness and generality.

Data-Availability Statement

Our artifact is available [26].

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable feedback on our paper and the anonymous artifact reviewers for their constructive suggestions. This work is sponsored by the National Natural Science Foundation of China under Grant No. W2411051.

References

[1] Anthropic. 2025. Claude Code. https://claude.com/product/claude-code.

[2] Anthropic. 2026. Claude-Sonnet-4.6. https://www.anthropic.com/news/claude-sonnet-4-6.

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

[4] Jinbao Chen, Hongjing Xiang, Luhao Li, Yu Zhang, Boyao Ding, and Qingwei Li. 2024. Utilizing Precise and Complete Code Context to Guide LLM in Automatic False Positive Mitigation. *CoRR* abs/2411.03079 (2024). doi:10.48550/ARXIV.2411.03079 arXiv:2411.03079

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

[6] Xueying Du, Kai Yu, Chong Wang, Yi Zou, Wentai Deng, Zuoyu Ou, Xin Peng, Lingming Zhang, and Yiling Lou. 2025. Minimizing False Positives in Static Bug Detection via LLM-Enhanced Path Feasibility Analysis. *CoRR* abs/2506.10322 (2025). doi:10.48550/ARXIV.2506.10322 arXiv:2506.10322

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

[8] 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, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 561–575. doi:10.1145/3314221.3314616

[9] Mohsen Iranmanesh, Sina Moradi Sabet, Sina Marefat, Ali Javidi Ghasr, Allison Wilson, Iman Sharafaldin, and Mohammad A. Tayebi. 2025. ZeroFalse: Improving Precision in Static Analysis with LLMs. *CoRR* abs/2510.02534 (2025). doi:10.48550/ARXIV.2510.02534 arXiv:2510.02534

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

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

[12] Tianchi Li and Xin Zhang. 2025. Combining Formal and Informal Information in Bayesian Program Analysis via Soft Evidences. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 143 (April 2025), 28 pages. doi:10.1145/3720508

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

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

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

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

[17] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and implementation of sparse global analyses for C-like languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 229–238. doi:10.1145/2254064.2254092

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

[19] OpenAI. 2025. Codex. https://openai.com/codex/.

[20] OpenAI. 2025. GPT-5.1-Codex-Max. https://openai.com/index/gpt-5-1-codex-max/.

[21] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided program reasoning using Bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and*

Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 722–735. doi:10.1145/3192366.3192417

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

[23] Yuanfeng Shi, Yifan Zhang, and Xin Zhang. 2025. On Abstraction Refinement for Bayesian Program Analysis. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 3232–3258. doi:10.1145/3763166

[24] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 265–266. doi:10.1145/2892208.2892235

[25] Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Cong Tian. 2024. Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far Are We? *ACM Trans. Knowl. Discov. Data* 18, 7 (2024), 168. doi:10.1145/3653718

[26] Yifan Zhang, Yuanfeng Shi, Haoran Lin, Yingfei Xiong, and Xin Zhang. 2026. *LLM-Based Alarm Resolution Guided by Bayesian Program Analysis (Paper Artifact)*. doi:10.5281/zenodo.21544166

[27] Yifan Zhang, Yuanfeng Shi, and Xin Zhang. 2024. Learning Abstraction Selection for Bayesian Program Analysis. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 954–982. doi:10.1145/3649845

[28] Yifan Zhang and Xin Zhang. 2026. Fuzzing Guided by Bayesian Program Analysis. *Proc. ACM Program. Lang.* 10, POPL (2026), 476–506. doi:10.1145/3776659

Received 2026-03-17; accepted 2026-08-06