Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

LLM-enhanced probabilistic modeling for effective static analysis alarms*

Xinlong PAN[†1,2], Jianhua LI[†‡1,2], Zhihong ZHOU[†‡1,2], Gaolei LI[1,2], Xiuzhen CHEN[1,2], Jin MA[1,2], Jun WU[1,2], Quanhai ZHANG[1,2]

[1]*Institute of Cyber Security and Technology, School of Computer Science, Shanghai Jiaotong University, Shanghai 200240, China*

[2]*Shanghai Key Laboratory of Information Security Integrated Management Technology Research, Shanghai 200240, China*

[†]E-mail: mr.p332@sjtu.edu.cn; Lijh888@sjtu.edu.cn; zhouzhihong@sjtu.edu.cn

Received Jan. 16, 2025; Revision accepted Apr. 14, 2025; Crosschecked

Unedited

Abstract: Static analysis presents significant challenges in alarm handling, where probabilistic models and alarm prioritization are essential methods for addressing these issues. These models prioritize alarms based on user feedback, thereby alleviating the burden on users to manually inspect alarms. However, they often encounter limitations related to efficiency and issues such as false generalization. While learning-based approaches have demonstrated promise, they typically incur high training costs and are constrained by the predefined structures of existing models. Moreover, the integration of large language models (LLMs) in static analysis has yet to reach its full potential, often resulting in lower accuracy rates in vulnerability identification.To tackle these challenges, we introduce BinLLM, a novel framework that harnesses the generalization capabilities of LLMs to enhance alarm probability models through rule learning. Our approach integrates LLM-derived abstract rules into the probabilistic model, utilizing alarm paths and critical statements from static analysis. This integration enhances the model's reasoning capabilities, improving its effectiveness in prioritizing genuine bugs while mitigating false generalizations. We evaluated BinLLM on a suite of C programs and observed 40.1% and 9.4% reduction in the number of checks required for alarm verification compared to two state-of-the-art baselines, underscoring the potential of combining LLMs with static analysis to improve alarm management.

Key words: Static analysis; Bayesian inference; large language model(LLM); Alarm ranking
https://doi.org/10.1631/FITEE.2500038 **CLC number:**

1 Introduction

Static analysis is a powerful technique for detecting bugs in software systems, yet it faces significant challenges in managing alarms effectively(Beller et al., 2016; Christakis and Bird, 2016; Muske and Serebrenik, 2022). A key post-processing method is user-guided ranking (Shen et al., 2011; Mangal et al., 2015), which prioritizes alarms based on user feedback. Bayesian program analysis formalizes this into a probabilistic framework: alarms are ranked by inferred probabilities, and user feedback iteratively refines the model (Raghothaman et al., 2018; Heo et al., 2019; Chen et al., 2021b; Kim et al., 2022; Zhang et al., 2024). Its effectiveness depends on the generalization ability of the Bayesian network, derived from inference graphs built using Datalog rules. More accurate rules yield stronger generalization, reducing user workload and improving alarm prioritization. However, this approach still relies on user expertise, and Datalog rules lack intelligent optimization (Muske and Serebrenik, 2022).

‡ Corresponding authors
* Project supported by the National Natural Science Foundation of China (Nos. U20B2048 and 62471301)
ORCID: Xinlong PAN, https://orcid.org/0009-0006-3328-4080
© Zhejiang University Press 2025

The integration of large language models (LLMs) into static analysis has gained attention (Chen et al., 2021a; Ma et al., 2023; Sun et al., 2024). However challenges such as hallucinations and randomness persist (Ji et al., 2023; Touvron et al., 2023), limiting their effectiveness in vulnerability detection. The state-of-the-art methods based on large language models have improved accuracy to 67.6% (Zhou et al., 2024), but these methods are constrained by data processing granularity, typically limited to the function or line level, which hampers their ability to detect vulnerabilities across entire libraries. Existing methods mainly use LLMs as judgment substitutes or static analysis supplements (Gao et al., 2023; Mohajer et al., 2023; Li et al., 2024a,b), lacking deep integration. This suggests a paradigm shift where LLMs actively enhance static analysis for improved accuracy (Li et al., 2024a).

To address these challenges, we propose BinLLM, a framework that leverages LLMs to refine alarm probability models through rule learning. By integrating LLM-derived rules with static analysis insights, including alarm paths and critical statements, our approach enhances reasoning capabilities and reduces user intervention, fostering a more synergistic relationship between LLMs and static analysis.

Unedited

2 Overview

We present our approach BinLLM in Fig. 1, leveraging LLMs to enhance real-time alarm probability modeling and efficiently identify true alarms. The approach consists of three stages: (1) A static analyzer extracts alarms and constructs an initial Bayesian network using Datalog rules. (2) The Bayesian network infers high-probability alarms, and the Alarm Path Extractor extracts alarm paths, and provides code slices to LLMs for validation and rule refinement. (3) The refined rules update the Bayesian network, iteratively improving alarm identification until convergence.

The static analyzer extracts syntactic feature tuples to define initial reachable edges. By applying path compression, pruning, and delooping from Bingo (Raghothaman et al., 2018), we transform the graph into a Bayesian network that is iteratively updated along the network edges.

To retain syntactic and semantic information while reducing redundancy, the Alarm Path Extractor refines alarm code by extracting relevant intra- and inter-program statements using Backward Program Slicing, preserving critical control and data dependencies at the line level.

However, using LLMs for code analysis will inevitably encounter problems such as hallucination, randomness (Ji et al., 2023; Touvron et al., 2023), and context window limitations. To address these issues, we adopt task decomposition (Wei et al., 2022; Li et al., 2024a; Yao et al., 2023), dividing the process into three stages: alarm characteristic identification, boundary estimation with key statement extraction, and rule learning. This ensures manageable context lengths and enhances LLM performance on specific tasks.

2.1 Contributions

This paper makes the following contributions:

1. We propose a learning-based probability model for static analysis alarms that leverages LLMs, utilizing alarm paths and key statements.
2. We introduce a method for extracting alarm paths through backward program slicing, enabling the identification of critical statements that influence alarm validity.
3. We evaluate our approach on a suite of C programs and compare it against two state-of-the-art alarm probability models, achieving performance improvements of 40.1% and 9.4%, respectively.

3 Semantic-based alarm path extractor

The primary limitation of the alarm probability model based on user feedback lies in its reliance on manually defined initial Datalog rules, which fail to effectively capture the syntactic and semantic features of complex alarms. This limitation hampers the model's ability to differentiate between related alarms. Consequently, it is essential to develop a method capable of extracting the syntactic and semantic characteristics of alarms to help refine and enhance the Datalog rules accordingly.

We have observed that the syntactic and semantic features of the code are the focus of identifying vulnerabilities. The semantic-based alarm path ex-

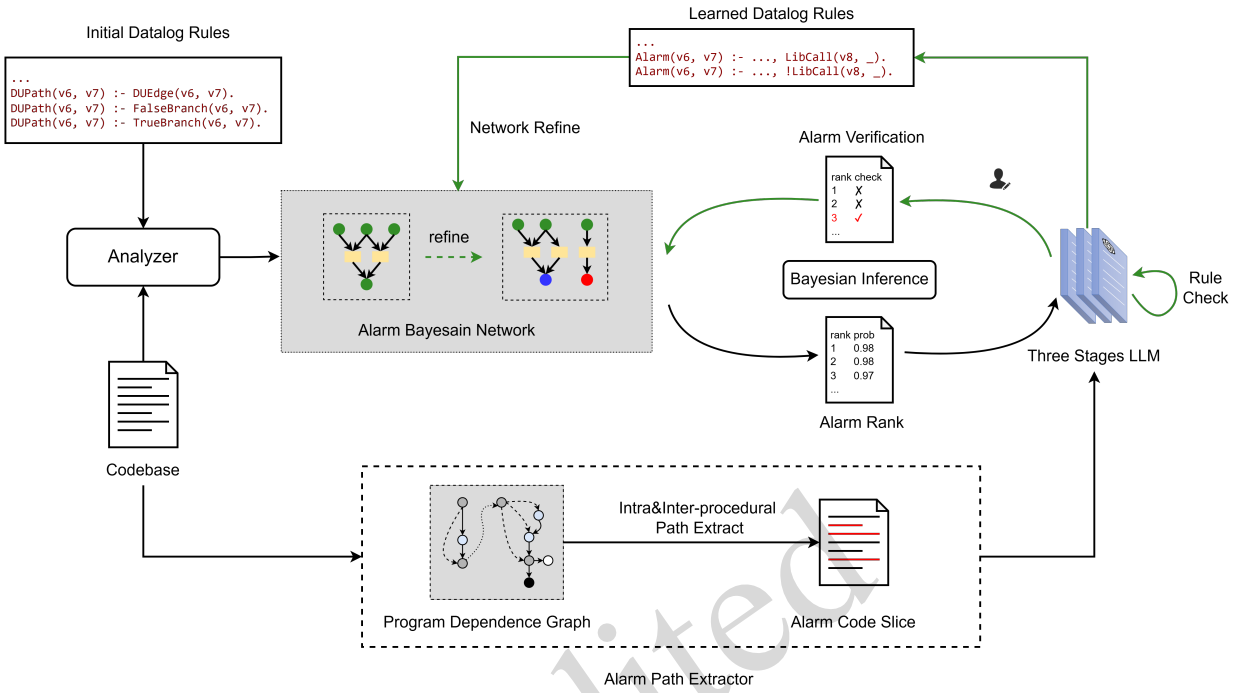

Fig. 1 Framework

Unedited

tractor utilizes both syntactic and semantic information, ouputing it at the granularity of code lines. The extracted alarm paths are subsequently provided as input to LLMs for vulnerability identification.

3.1 Basic definition

In order to take advantage of the semantic features of the source code, we use intra-program slicing and inter-program slicing to extract alarm paths. For this we need a program dependency graph(PDG), including data dependencies and control dependencies. Let's clarify these definitions below.

Definition 1 (AST). Given a program $P = \{f_1, \ldots, f_\eta\}$, the abstract syntax tree (AST) of a function f_i is denoted as $T_i = (N_i, E_i)$, where $N_i = \{n_{i,1}, \ldots, n_{i,c_i}\}$ is the set of nodes, each representing a statement $s_{i,j}$ or token $t_{i,j}$. The edges $E_i = \{e_{i,1}, \ldots, e_{i,d_i}\}$ define the hierarchical syntax structure, where tokens represent operators, variable names, or keywords, and the parent-child relationships encode the syntactic organization.

Definition 2 (PDG). (Ferrante et al., 1987) The PDG integrates data and control dependencies within a function f_i of a program $P = \{f_1, \ldots, f_\eta\}$. It is denoted as $G_i''' = (V_i, E_i''')$, where V_i represents statements or control points, and $E_i''' = E_i' \cup E_i''$ combines data dependency edges E_i' and control dependency edges E_i''. The data dependency graph (DDG) captures data flow between statements, while the control dependency graph (CDG) models execution dependencies based on post-dominance in the control flow graph. The PDG provides a unified representation of both dependencies.

3.2 Backward program slice

After obtaining the AST and PDG, we can perform backward program analysis to obtain a backward program slice related to the alarm code. The reason for adopting backward program analysis instead of forward program analysis is that we are only concerned about whether the data flow where the alarm is located will have out-of-bounds, illegal access, and other situations, while the data flow after the alarm is not our analysis target; compared with starting from the entry function and performing forward analysis until reaching the code where the alarm is located, starting backward analysis from the alarm code is more efficient and can avoid a large number of invalid exploration paths (Khanfar et al., 2015; Lisper et al., 2015).

We divide the Backward Program Slice into two parts: intra-procedural backward slice and inter-

```
1:  static char *suffixes[] = { "gz", "bz2", "xz", "zip",
NULL };
2:
3:  void strip_compression_suffix(const char *name)
4:  {
5:   char *suf = strrchr(name, '.');
6:   if(suf)
7:   {
8:     int suffix_len = strlen(++suf), name_len = 0;
9:     for (int i = 0; compression_suffixes[i]; i++)
10:    {
11:     if (strlen(suffixes[i]) == suffix_len)
12:     {
13:      name_len = strlen(name) - suffix_len-1;
14:      // ...
15:      break;
16:     }
17:    }
18:    if (name_len > 4 &&
           !strncmp(name + name_len - 4, ".tar", 4))
19:     name_len -= 4;
20:    // ...
21:    }
22:    // ....
23: }
```

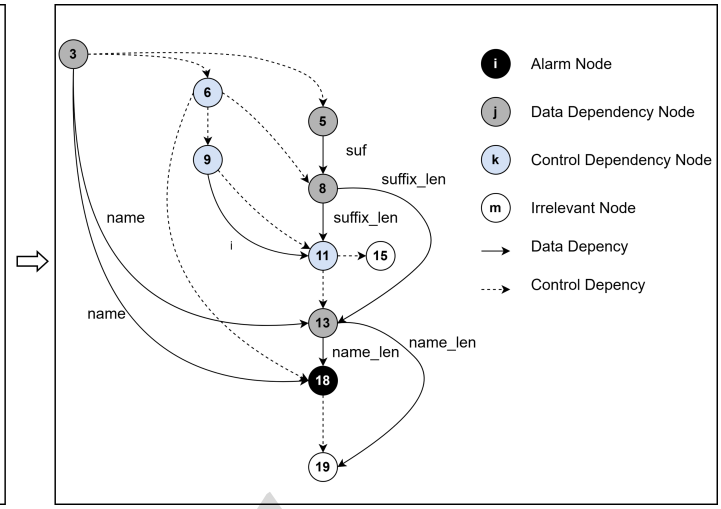

Fig. 2 Example of intra-procedural backward slice

Unedited

Algorithm 1: Intra-program Slicing

Input: Function name f, alarm call ID id line number l

Output: Variable dictionary V, list of slice lines L

Function `intraProgramSlice`(f, id, l)**:**
- $Q \leftarrow \{\text{vars}\}$;
- $V \leftarrow \emptyset$;
- $L \leftarrow \emptyset$;
- **while** $Q \neq \emptyset$ **do**
  - $x, l, id \leftarrow Q.pop()$;
  - $S_{define} \leftarrow$ getDefineStatements(f, x, l, id);
  - **foreach** $m \in S_{define}$ **do**
    - $D_{data} \leftarrow$ extractDDG(m, x);
    - $V_x \leftarrow V_x \cup \{(var, l', id') \in D_{data}\}$;
    - $D_{control} \leftarrow$ extractCDG(m);
    - $V_x \leftarrow V_x \cup \{(s, l'', id'') \in D_{control}\}$;
    - $Q \leftarrow Q \cup \{(var, l', id') \in D_{data}\}$;
  - $V \leftarrow V \cup V_x$;
- $L \leftarrow$ joinControlStructures$(f$, V$)$;
- **return** V, sort(L);

Algorithm 2: Backwards Inter-procedural Program Slice

Input: Function name f, call ID id

Output: Slices across functions C

Function `backwardCallPath`(f, id, T)**:**
- **if** $(f, id) \in T$ **then**
  - Return $\{T\}$;
- $T \leftarrow \{(f, id)\} \cup T$;
- $C \leftarrow$ `getCallers`(f, i);
- $P \leftarrow \emptyset$;
- **foreach** $(f', id') \in C$ **do**
  - $P_{f'} \leftarrow$ `backwardCallPath`(f', id', T);
  - **foreach** $p \in P_{f'}$ **do**
    - Add p to P;
- Return P;

Function `interProgramSlice`(f, id)**:**
- $C \leftarrow \emptyset$;
- $P \leftarrow$ `backwardCallPath`$(f, id, \emptyset)$;
- $p \leftarrow$ `getHighestWeightPath`(P);
- **foreach** $(f, id) \in p$ **do**
  - $l \leftarrow$ `getLine(f,id)`;
  - $L_f \leftarrow$ `intraProgramSlice`(f, id, l);
  - $C \leftarrow (f, L_f \cup C)$;
- Return C;

procedural backward slice. Intra-procedural backward analysis outputs the code within the function that has data dependency or control dependency with the alarm code. Inter-procedural backward analysis locates the call path of the function where the alarm is located, and outputs the most likely/-most frequently used call path from the entry function (if found) to the alarm function. It performs intra-procedural analysis at each call site to form a complete program slice.

3.2.1 Intra-procedural backward slice

Algorithm 1 demonstrates the complete process of program slicing within a procedure. It outputs line slices for specified functions and statements based on data dependencies and control dependencies. Next, we will elaborate in detail with the example shown in Fig. 2. The code in Fig. 2 comes from real-world code (tar-1.28) containing a buffer overflow vulnerability. At line 18, `strncmp(name + name_len - 4, ".tar", 4)` can cause an out-of-bounds access when `name_len<4`. Fig. 2 shows the slicing process of Algorithm 1 for this alarm code.

Step 1 (Lines 2–4): Initialize the queue of variables to be processed Q, the variable dictionary (containing variable names and corresponding slice line numbers) V, and the slice line numbers L. In the above example, Q is initialized to `{name, name_len}`.

Step 2 (Lines 6–7): Obtain the current variable x, line number l, and statement id from Q. Also, retrieve the define statements in the data flow graph that directly reach id before l. Although the granularity here is at the line level, the actual analysis process is still at the statement level, but the analysis results of statements on the same line are merged together. When x=`name`, its define statement is the statement at $line3, 18$, which is the function entry, indicating that it comes from parameter passing. Note that l is also returned as a result because a line of code may contain multiple statements, some of which may also include data dependencies or control dependencies.

Step 3 (Lines 8–14): Sequentially extract the data dependencies and control dependencies contained in the PDG, and add the variables found in the data dependencies to the queue of variables to be analyzed Q, as shown in Fig. 2, where the variable `suffix_len` at $line13$ will be added to Q as the object of subsequent slicing. Finally, add the slice information found about x to V, including related variables and line numbers.

Step4 (Lines 15–16): Since we need to create a complete program slice, we need to merge the control flow and complete the statements. Finally, return the slicing result, including the involved variables V and the corresponding line numbers L.

For the example in Fig. 2, after running Algorithm 1, we will obtain a slice from data dependencies `{3,5,8,13,18}`, and a slice from control dependencies `{6,9,11}`. It is worth noting that we did not perform further data flow analysis on statements with function calls. Instead we place them in the LLMs recognition stage, allowing the LLMs to decide for themselves whether they need the definitions of these functions. This is because, on the one hand, not all functions can have their data flow and control flow obtained in the static analysis stage, such as library functions and dynamically linked code. On the other hand, not all called functions in the slice need to be traced back to their definitions. Some function calls do not affect the alarm code, and some function calls can be inferred from the context. Therefore, we leave this task to the LLMs to judge whether it needs the definition of a specific function or type.

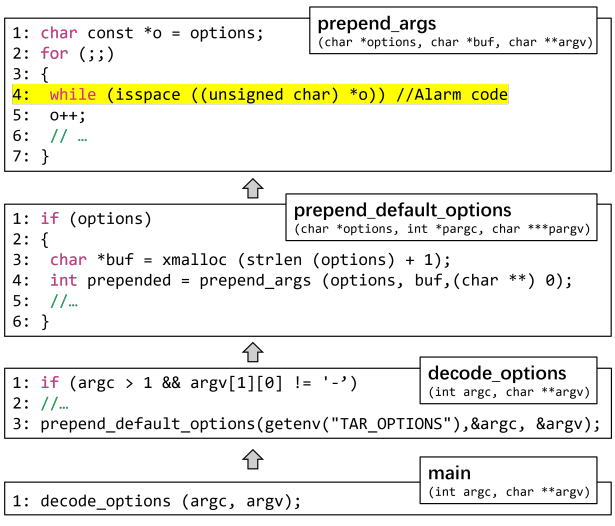

Fig. 3 Inter-procedural backward slice example

3.2.2 Inter-procedural backward slice

In certain scenarios, relying solely on intra-procedural slicing may not yield an accurate determination of the veracity of an alarm,

Unedited

as illustrated in Fig. 3. The function `prepend_args` contains alarm code at Line 4: `while (isspace ((unsigned char) *o))`. When `o` is a null pointer, the dereference operation `*o` will result in a null pointer dereference issue. If only considering intra-procedural slicing, this alarm would be considered a True Alarm. However, the function `prepend_args` only exists in one call path throughout the entire program, as shown in Fig. 3: `main` → `decode_opetions` → `prepend_default_options` → `prepend_args`. Within this call path, the condition where `options` is a null pointer has already been checked; hence, this alarm would not be triggered during actual execution.

Algorithm 2 shows the complete process of program slicing crossing functions.The process is divided into two parts: `backwardCallPath` uses DFS to find all call paths that reach the function f where the alarm is located. `interProgramSlice` selects the path with the highest number of calls as the target, performs intra-procedural function slicing on each call site, and then concatenates them in the order of calls to form an inter-procedural function slice of the alarm.

Similar to intra-procedural slicing, Backward analysis is also used here to find all callers of the alarm function, which serve as parent nodes in its call graph. This process continues recursively until the entry function is found or a cyclic call is encountered.

In real-world code, the call graph can be very large, and the number of call paths from the entry function to the alarm function can be considerable. However, in the subsequent LLMs recognition part, due to the window limitation of the LLMs, the input content should be as concise and critical as possible. Based on this, we propose a frequency-based call path selection method.

$$\begin{aligned} P_{\max} &= \arg\max_{P_k} \left(W(P_k)\right) \\ &= \arg\max_{P_k} \left(\frac{\sum_{i=1}^{m} w(f_i^k)}{m}\right) \end{aligned} \tag{1}$$

Given a set of call paths $P_1, P_2, \ldots, P_n$, path P_k contains several functions $f_1^k, f_2^k, \ldots, f_m^k$, where m is the length of the path. The weight of each function f_i^k is represented by $w(f_i^k)$, which equals its number of calls $c(f_i^k)$. The average weight $W(P_k)$ of path P_k is calculated as the sum of the weights of all functions in the path divided by the length m, where $w(f_i^k)$ is the weight of function f_i^k, equal to its number of calls $c(f_i^k)$.

The call path obtained from Formula (1) is the one with the highest average number of calls among all call paths. On the one hand, this ensures that $P_{\max}$ is the most likely path to be called; on the other hand, it imposes a constraint on the path length. Since the call path output by Algorithm 2 does not contain cyclic calls, call paths that are too long do not have an advantage in terms of average number of calls.

4 LLM alarm identification and rule learning

To optimize the manual aspects of code static analysis alarm identification, we have introduced LLMs for auxiliary analysis(Ahmed et al., 2023; Pearce et al., 2023; Pei et al., 2023) and rule learning. Utilizing the semantically-based inter-program alarm code slicing provided by the vulnerability exploitation path extractor introduced earlier, the LLMs will provide a judgment on the feasibility of the alarm occurrence, including buffer overflow and pointer dereference vulnerabilities under address calculation and boundary judgment. In view of the LLMs' performance issues in complex problems (Achiam et al., 2023) and considering cost issues, we use prompt engineering to improve the LLMs' performance in identifying static analysis alarms.

The decision to employ prompt engineering instead of alternatives, such as model fine-tuning or specialized code models like CodeBERT(Feng et al., 2020), is based on several considerations. First, specialized code models may have already learned from benchmark datasets, potentially biasing their outputs and undermining the evaluation of the LLMs' ability to independently analyze and learn from alarm contexts. Second, model fine-tuning requires high-quality labeled samples specific to the task. However, our approach evaluates alarms across entire codebases, making it challenging to construct a sufficiently annotated dataset. Additionally, the scale of the involved code data is prohibitively large for efficient fine-tuning. Third, prompt engineering is more cost-effective and scalable, offering adaptability across diverse datasets without being overly dependent on specific training sets. This flexibility

Unedited

ensures the robustness of the analysis and facilitates broader application scenarios.

We divide the entire prompt and interaction process with LLMs into three sub-stages: identification of alarm type, statements, and key variables; boundary estimation and identification of key exploitation paths for alarms; and optimization of Datalog rule output. The conclusions of each stage will serve as the input for the next stage, allowing the LLMs to focus on one task in each round of dialogue and maintain a relatively controllable context window length.

4.1 Prompt design

The vulnerabilities we analyze include buffer overflow and pointer dereference issues, both of which have great similarities in identification, with the primary challenge being the determination of address validity. Therefore, in our design, we group these two types of alarms together for identification, allowing the LLMs to automatically distinguish between them.

Our prompt design is divided into three phases. The output of each phase serves as the input for the subsequent phase, ensuring that the content at each stage is kept to a minimum. This approach allows the LLMs to focus more effectively while reducing contextual overhead.

- **Phase One: Type, Statement, and Variable Identification**: Given the alarm code and slicing information within the function, this phase aims to locate the alarm call statements, including parameters for memory operation functions and pointer variables.

- **Phase Two: Boundary Estimation and Key Statement Identification**: The alarm path extractor provides a complete program slice. Based on the identification results from Phase One, the LLMs perform overflow calculations or pointer offset calculations for different types of alarms. They also determine the key statements that impact the veracity of the alarms. For example, in Fig. 3, the key statements affecting the null pointer dereference involve the null check for the `options` variable and the while loop in which the alarm resides.

- **Phase Three: Rule Learning**: This phase involves reading the output from Phase Two and converting the key alarm statements into corresponding rules in a format supported by Datalog. Currently, the supported syntax nodes include various alarm types, operators, control flow statements, and more. Each alarm learning iteration will output multiple rules, which will then be input into the alarm Bayesian network to facilitate structural optimization.

4.2 Rule refine

The Datalog rules generated in Phase Three(Rule Learning) will be fed back into the Bayesian network construction phase. Utilizing the learned alarm path information, the new rules may encompass previously unencountered types of nodes, such as loop heads, library function calls, and assignments. In conjunction with the output from the static analysis engine, Sparrow(Oh et al., 2012), we provide a total of 72 types of statement nodes for the LLMs to match and generate corresponding Datalog rule variables. These rules, which reflect the actual alarm paths, will enhance the ability to distinguish the authenticity of alarms more effectively.

However, due to the inherent randomness of large models, there is no guarantee that every output will meet the input requirements for Datalog. Some errors may lead to premature termination of Datalog inference, while others may cause the inference process to run indefinitely.

Examples of such issues include:

- **Ungrounded variable**(Sou):

  $$\text{Alarm}(v6, v7)\text{:-...}, \text{S}(v6, v7, v8), !\text{LibCall}(v8, v9)$$

  The variable v9 is not bound as an argument of a positive predicate in the body.To satisfy Datalog's requirements, the rule should be modified to $!\text{LibCall}(v8, _)$;

- **Invalid tuple**:

  $$\text{Alarm}(v6, v7)\text{:-...}, \text{S}(v6, v7, _), \text{Alloc}(v8, _, _)$$

  The presence of the tuple $\text{Alloc}(v8, _, _)$ is irrelevant to the other tuples in the rule. Since it does not contribute to the logical conclusion of the alarm condition, this results in an invalid tuple.

Unedited

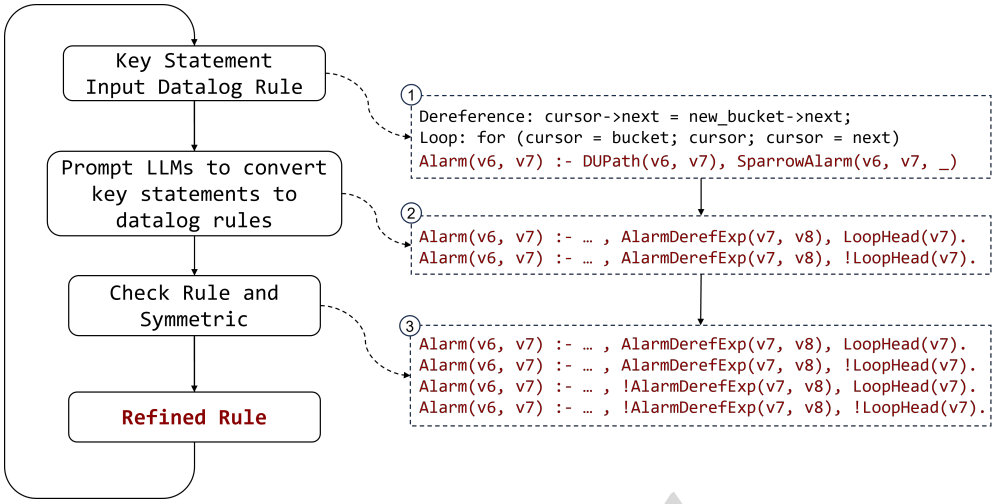

Fig. 4 LLMs rule refine design

Unedited

To address the uncertainty in the output rules generated by LLMs and to reduce the costs associated with manual checks, we propose the following components to tackle these issues. The overall framework is illustrated in Fig. 4.

1. **Rule expansion based on input**: We initiate the expansion process with the simplest initial rules, allowing LLMs to add new premise tuples and instantiate tuple variables without altering the conclusions or existing premises. This facilitates the mapping of alarm paths to rules. In subsequent learning iterations, the optimized rules will be provided as input to the model, ensuring that the rules generated by the LLMs can progressively delve into more complex rule learning over multiple iterations.

2. **Symmetric rules**: For each new premise added, there must be a corresponding rule that maintains the structural integrity of the foundational rule, differing only in the introduction of a negated premise. For example, if an advanced rule introduces a new premise, there should also be a rule that negates this premise. If two premises are added, four variations must exist to ensure comprehensive coverage of both positive and negative scenarios while preserving the logical conclusions. This method is inspired by the learning principles of BayeSmith(Kim et al., 2022) , where the addition of a pair of symmetric premises ensures that the conclusion remains unchanged, while providing more detailed reasoning. In the following rules, regardless of whether the $v7$ node is a library function call, the conclusion that $v6$ and $v7$ are alarm nodes can be drawn; however, the paths of reasoning differ. This variation allows for the assignment of different probabilities to these rules within the Bayesian network, enabling the distinction of probability calculations for alarm nodes under different circumstances.

$$\text{Alarm}(v6, v7) \text{:-} \ldots, \text{Alloc}(v7, _), \text{Libcall}(v7, _).$$

$$\text{Alarm}(v6, v7) \text{:-} \ldots, \text{Alloc}(v7, _), !\text{Libcall}(v7, _).$$

3. **Rule checking**: To ensure that the outputs of the LLMs do not exhibit issues such as ungrounded variables and invalid tuples, we implement **task decomposition** and **self-validation** to standardize the rule outputs. The rule learning process in Stage Three consists of two steps: first, generating corresponding rules based on the earlier analysis results; and second, modifying these rules in accordance with Datalog specifications. Additionally, we incorporate a self-validation step, wherein the LLMs is required to review and correct its own outputs. Finally, we have developed a rule checker using Python; if any errors are still detected in the rules, the process will undergo further iterations for refinement.

4.3 Bayesian network learning

4.3.1 Case analysis of the Datalog rule refine

In Listing 1, we present a simplified version of the real vulnerability code from tar-1.28 corresponding to line 7. This code, which is consistent with the code shown in Fig. 2, has been simplified to include a potential false alarm corresponding to line 11 as identified by static analysis. After processing by the Datalog engine, the initial Datalog derivation graph, shown in Fig. 5, is produced. In this graph, gray nodes represent input tuple nodes (fact nodes), white nodes denote output tuple nodes derived through rule inference, and the two alarm nodes, `Alarm(7)` and `Alarm(11)`, are marked with double borders. `Alarm(7)` corresponds to the true alarm at line 7, while `Alarm(11)` represents a false alarm at line 11.

Listing 1 A vulnerable code segment from tar-1.28

```
1  char *strip_compression_suffix(const
       char *name)
2  {
3   char *s = NULL;
4   size_t len;
5   if(find_compression_suffix(name,&len))
6   {
7    if(strncmp(name+len-4,".tar",4)==0)
8     len -= 4;
9    // ...
10   s = xmalloc (len + 1);
11   memcpy (s, name, len);
12   // ...
13  }
14  return s;
15 }
```

Due to the overly basic nature of the initial Datalog rules, the original derivation graph fails to distinguish between `Alarm(7)` and `Alarm(11)`, which impedes effective detection of the true alarm. Specifically, if the network identifies `Alarm(11)` as a false alarm before identifying `Alarm(7)`, this will propagate along `DUPath(5,7)` and update the probability of `Alarm(7)`, causing the true alarm to be ranked lower—a phenomenon known as false generalization, which will be discussed in the next subsection.

By leveraging LLM-based learning, BinLLM refines the Datalog rules based on the actual alarm context, effectively resolving this issue. In the case of `Alarm(7)`, the vulnerability is caused by an improper computation in the expression `name+len-4`. From this expression, BinLLM extracts the relation `PlusPI` (denoting a pointer arithmetic involving an integer offset). Using the rule refinement method, BinLLM generates the following Datalog rules:

$$\text{DUPath}(v6, v7)\text{:-DUEdge}(v6, v7), \text{PlusPI}(v7).$$

$$\text{DUPath}(v6, v7)\text{:-DUEdge}(v6, v7), !\text{PlusPI}(v7).$$

In this context, `DUEdge(v6, v7)` and `DUPath(v6, v7)` denote the presence of direct data flow and transitive data flow, respectively, from code location $v6$ to $v7$. Moreover, `PlusPI(v7)` indicates that a pointer arithmetic operation involving an integer offset occurs at $v7$, as exemplified by the expression `name+len-4` in this case.

Reconstructing the Datalog derivation graph with these refined rules yields the graph shown in the right half of Fig. 5. The introduction of the new node `PlusPI(7)` facilitates a clear distinction between `Alarm(7)` and `Alarm(11)`. By differentiating `DUPath(5,7)` based on the presence or absence of `PlusPI`, the negative impact of the false alarm `Alarm(11)` on the probability computation of the true alarm `Alarm(7)` is effectively mitigated, leading to more accurate alarm identification.

4.3.2 False generalization in Bayesian program analysis

Table 1 Rank change in false generalization(Kim et al., 2022)

Rank	Alarm	Prob.
1	Alarm(7)	0.93
2	Alarm(11)	0.92
...		
9	Alarm(10)	0.91
...		

→

Rank	Alarm	Prob.
...		
...		
461	Alarm(11)	0.41
462	Alarm(10)	0.40
...		

Bayesian program analysis uses user-interactive alarm feedback results as posterior knowledge to update the probabilities of other alarms, a process referred to as generalization. A Bayesian network with stronger generalization capabilities can identify more true alarms with fewer interactions. However, in some cases, feedback may erroneously lower the ranking of true alarms, as shown in Table 1; this phenomenon is known as false generalization(Kim et al., 2022). The root cause of false generalization lies in the Bayesian network's inability to differentiate the features of various alarms, leading to highly correlated probabilities of related alarm nodes derived

Unedited

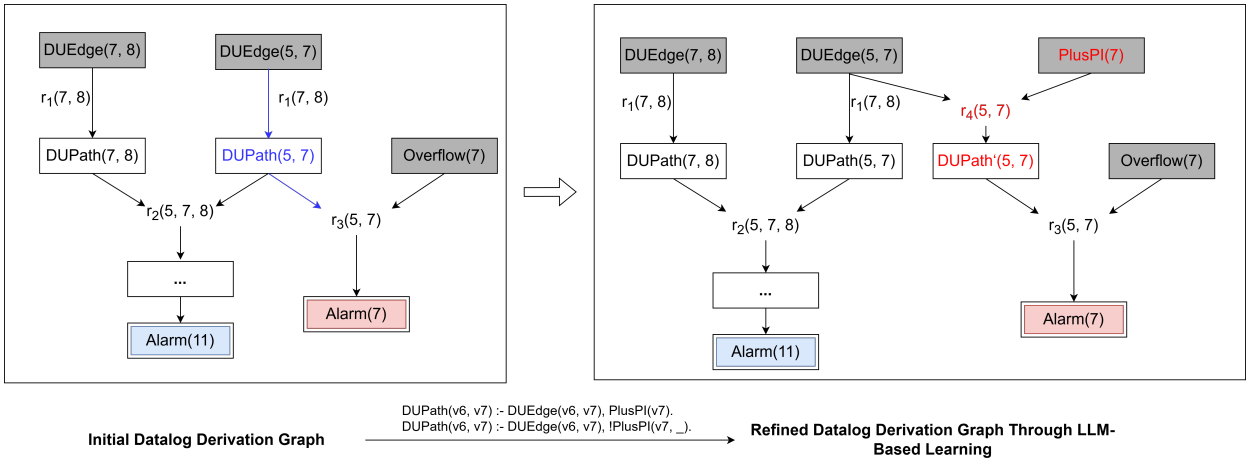

Fig. 5 Datalog-rule-based refine workflow for case in Listing 1

from the same rules. For instance, certain alarms may critically rely on loop constructs, yet the rules used to build the Bayesian network do not recognize these loops, making it impossible to distinguish alarms that involve loops from those that do not.

Unlike BayeSmith(Kim et al., 2022), our learning approach does not aim to directly address false generalization. Instead, by targeting the root causes of false generalization, our method effectively prevents the decline in the ranking of true alarms caused by alarm extraction rules. Furthermore, our learning process does not depend on the current false generalization situation. It does not require additional library code as a training set to optimize the Bayesian network. Instead, it directly learns the key factors that constitute the alarm and extracts corresponding rules to optimize the performance of the probabilistic model. Detailed parameter settings and the effectiveness of our approach in mitigating False Generalizations can be found in the experimental results section.

5 Experimental evaluation

Our experiments were conducted on a machine running Ubuntu 16.04, equipped with an Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz.

5.1 Setting

5.1.1 Dataset

We utilize Sparrow(Oh et al., 2012) as the static analysis framework for the C language and Lib-DAI(Mooij, 2010) as the Bayesian network inference framework. As shown in Table 2, the benchmark suite includes 14 C codebases. Our focus is primarily on buffer overflow and pointer dereference issues, with 8 out of the 14 benchmarks specifically consisting of these types of problems. The remaining six benchmarks target integer overflow and format string vulnerabilities to evaluate the scalability of our approach.

Table 2 Benchmark characteristics

Program	KLOC	#Bugs	Bug type	Reference
cflow-1.5	40	1	Buffer overrun	MITRE (2019a)
fribidi-1.0.7	13	1	Buffer overrun	MITRE (2019b)
grep-2.19	68	1	Buffer overrun	MITRE (2015a)
gzip-1.2.4a	9	14	Buffer overrun	Heo et al. (2017)
patch-2.7.1	51	1	Buffer overrun	MITRE (2016)
sort-7.2	98	1	Buffer overrun	Eggert (2010)
tar-1.28	112	1	Buffer overrun	Meyering (2018)
wget-1.12	65	6	Buffer overrun	Rühsen (2018a)
jhead-3.0.0	5	2	Integer overflow	Rühsen (2018b)
opting-0.5.3	61	1	Integer overflow	Heo et al. (2017)
autotrace-0.31.1	18	18	Integer overflow	MITRE (2017b)
urjtag-0.8	46	6	Format string	MITRE (2015b)
a2ps-4.14	64	6	Format string	MITRE (2017a)
sdop-0.61	23	65	Format string	MITRE (2018)

5.1.2 Baseline

We compare BinLLM against two baseline frameworks: Bingo(Raghothaman et al., 2018) and BayeSmith(Kim et al., 2022). Bingo is an interactive alarm-ranking system that leverages user feedback. BayeSmith builds on Bingo, aiming to address the issue of false generalization. Starting from an initial model and a set of training programs with bug labels, BayeSmith iteratively refines rules based on feedback to mitigate false generalization and improve overall effectiveness.

Unedited

5.1.3 LLM setting

Our experiments were conducted on GPT-4o, accessed via OpenAI's API (version gpt-4o-0806). In each learning iteration, the LLMs receive input from the top three highest-ranked alarms, after which they output optimized rules and reconstruct the Bayesian network and alarm ranking. This process is repeated for five iterations. Afterward, we perform a full alarm identification phase without further rule interactions, continuously identifying the alarms with the highest current probability until all bugs are found.

5.2 Effectiveness

Table 3 LLM alarm key statement recognition metrics. Each code line is considered as a sample, and the line containing the key statement that affects the alarm is considered positive, otherwise it is negative. (C) indicates the subset of cases correctly identified by the model.

Program	TP(C)	TN(C)	Precision	Accuracy	Recall	F_1
Interval	16(14)	747(747)	1	0.997	0.874	0.933
Taint	23(19)	1007(1005)	0.905	0.994	0.826	0.864

In this section, we evaluate the effectiveness of alarm path extraction and recognition and probability model, and compare them with the baseline.

5.2.1 Effectiveness of alarm path extraction and recognition

Our slicing approach effectively condenses code while preserving alarm-relevant content. Intra-procedural slices average 48.0 lines, achieving a 72.0% compression from the original 171.4 lines, while inter-procedural slices reduce 624.8 lines to 139.9, with a 77.6% compression. This demonstrates significant pruning of irrelevant code, especially across functions. Additionally, the highest-weighted call path averages 3.7 in length, closely matching the original 4.1 (91.9%), ensuring accurate extraction with minimal interference. These results highlight the efficiency of slicing in refining BinLLM analysis.

Table 3 reports the performance of LLMs in identifying key sentences for two alarm types: Interval (representing Buffer Overrun analysis) and Taint (representing Integer Overflow and Format String analysis). The results are promising. Manual verification of 25 randomly selected LLM-generated analysis reports revealed high recall rates: 87.4% for Interval and 82.6% for Taint. Precision and accuracy exceed 90% for both types, attributed to the alarm path extractor's ability to eliminate irrelevant sentences and the incorporation of techniques such as task decomposition and progressive prompting, which simplify analysis steps.

However, some key statements were not correctly identified, primarily due to missing operation statements for certain variables. Alarm generation often involves interactions between multiple variables, but LLMs analyses occasionally focus on changes in only a single variable. In Taint analysis, the inability to fully recognize functions processing user input limits attention to source-variable changes, leading to slightly lower performance compared to Interval analysis. Despite these challenges, the approach demonstrates strong capabilities in isolating alarm-relevant statements and improving static analysis accuracy.

5.2.2 Effectiveness of probabilistic models

Table 4 presents the detection efficiency across benchmarks for Bingo, BayeSmith, and our method, BinLLM$_N$ andBinLLM. BinLLM$_N$ refers to BinLLM operating without the alarm path extractor. In this configuration, the source code of the function containing the alarm serves as the input for alarm information. This setup enables an effective evaluation of the alarm path extractor's impact on performance.

For BinLLM$_N$, its performance surpasses that of Bingo on the majority of datasets (9 out of 14). This indicates that even in the absence of the alarm path extractor, the LLMs can effectively learn alarm-related information and enhance the Bayesian probability model through the application of prompt-based techniques. However, compared to BayeSmith and BinLLM, its results fall short of expectations. Specifically, its performance is 27.7% lower than BayeSmith and 40.9% lower than BinLLM, highlighting its limitations. This disparity is primarily due to the fact that most alarms stem from interprocedural data flow or taint propagation, which cannot be fully captured through function-level analysis alone. Furthermore, the slicing mechanism provided by the alarm path extractor reduces the analysis scope for the LLMs, allowing it to concentrate more effectively on alarm-relevant

Unedited

statements.

For BinLLM, the results indicate that the Bayesian network constructed by BinLLM through rule learning outperforms Bingo on all datasets (with the exception of parity in the urjtag dataset) and surpasses BayeSmith in the most (10 of 14) datasets. In the case of urjtag, where BinLLM's performance aligns with Bingo, this is likely due to the dataset's already low false-positive rate, resulting in limited optimization potential.

However, in the wget-1.12 benchmark, BinLLM shows only a 1.0% improvement over Bingo, with a considerably higher iteration count than BayeSmith. We attribute this to two main factors: first, due to token limitations and cost considerations, we restricted each dataset to five learning iterations, with only three new alarms per iteration, totaling 15 alarms—an insufficient representation given the large number of alarms, potentially limiting BinLLM's ability to identify features associated with real bugs. Second, the inherent randomness in LLMs affects the reliability of rule-learning outputs, meaning each rule learned may not consistently target critical nodes for alarm differentiation. Although we ensured correctness in rules via expansion and symmetry, effectiveness could not be fully guaranteed.

Overall, while BinLLM shows limited improvement in a few cases, it provides an average detection efficiency increase of 40.1% over Bingo, 9.4% over BayeSmith and 29.0% over BinLLM$_N$. This indicates BinLLM's potential to significantly reduce the number of checks required in alarm ranking and the necessity of the alarm path extractor. Additionally, unlike BayeSmith, BinLLM does not require pre-labeled training programs, allowing direct learning on the target dataset with fewer constraints for practical applications.

5.3 Reduction of false generalization

BinLLM leverages alarm-specific rule learning to mitigate false generalization. We assess this effect by measuring the frequency and magnitude of false generalizations, as presented in Table 5. Similar to BayeSmith, we define a false generalization as a drop in rank for true alarms during an interaction. On average, BinLLM reduces the frequency of false generalizations by 17.8%, the magnitude by 41.4%, and the overall product of **Freq**×**Mag** by −0.05%.

Table 4 Effectiveness comparison. BINLLM$_N$ indicates BINLLM without the alarm path extractor.

Program	#Alarms	Bingo	BayeSmith	BinLLM$_N$	BinLLM
cflow-1.5	805	94	60	62	49
fribidi-1.0.7	213	6	3	5	3
grep-2.19	912	53	53	63	37
gzip-1.2.4a	358	145	107	170	125
patch-2.7.1	502	36	34	35	27
sort-7.2	715	176	94	176	15
tar-1.28	1369	218	146	88	86
wget-1.12	891	193	90	193	191
jhead-3.0.0	19	7	4	5	3
optipng-0.5.3	67	14	12	12	9
autotrace-0.31.1	77	77	43	49	34
urjtag-0.8	35	22	17	22	22
ap2s-4.14	27	15	11	7	7
sdop-0.61	150	85	81	77	76
Total	6140	1141	755	964	684

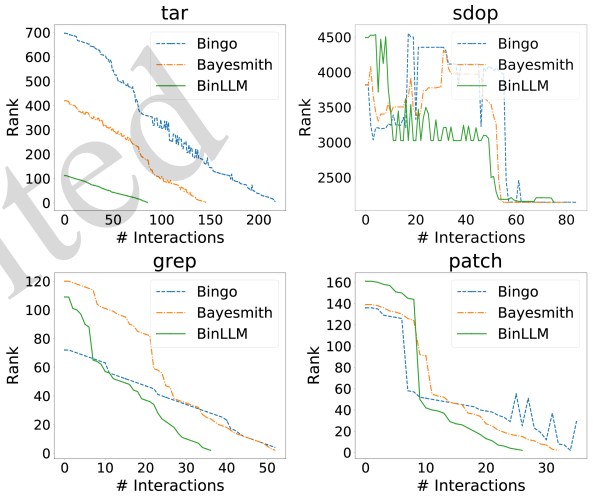

Fig. 6 Comparison of rank history. Each data point represents the sum of the rankings of all bugs.

Table 5 Reduction of false generalizations.Freq and Mag report the number of false generalizations and their average magnitude for the bayesian program analysis.

Program	Bingo			BinLLM		
	Freq	Mag	Freq×Mag	Freq	Mag	Freq×Mag
cflow-1.5	7	22.1	155.0	4	1.0	4.0
fribidi-1.0.7	2	2.0	4.0	1	2.0	2.0
grep-2.19	0	0.0	0.0	0	0.0	0.0
gzip-1.2.4a	191	15.3	2931.8	80	8.9	714.4
patch-2.7.1	4	26.0	104.0	0	0.0	0.0
sort-7.2	7	73.0	511.0	0	0.0	0.0
tar-1.28	27	19.4	523.0	0	0.0	0.0
wget-1.12	122	88.8	10839.7	94	88.9	8360.4
jhead-3.0.0	0	0.0	0.0	0	0.0	0.0
optipng-0.5.3	3	21.7	65.0	2	39.5	79.0
autotrace-0.31.1	194	12.1	2343.5	152	17.1	2597.7
urjtag-0.8	6	11.0	66.0	14	7.6	106.0
ap2s-4.14	8	4.1	33	1	1	1
sdop-0.61	1552	5.5	8598.1	1396	10.3	14323
Average	151.6	21.5	1869.6	124.6	12.6	1870.5

Fig. 6 illustrates the rank variation for true alarms over multiple interactions across four benchmarks.

From Table 5, it is evident that BinLLM reduces both the frequency and magnitude of false gen-

Unedited

eralizations in most cases.However, due to its poor performance on autotrace-0.31.1 and sdop-0.61, its overall level on **Freq**×**Mag** is slightly higher than BINGO.

In the ranking histories for benchmarks tar-1.28 and grep-2.19 in Fig. 6, BINLLM exhibits a smoother and more consistent decrease in rank compared to BINGO. Conversely, on sdop-0.61, the false generalization magnitude of BINLLM is higher than that of BINGO, resulting in an increase in its overall false generalization rate. This is primarily attributed to the fact that sdop-0.61 contains a significantly higher number of bugs, accounting for nearly half of the total alarm level (65 of 150). Nevertheless, BINLLM employs more effective rules that induce greater changes in the alarms associated with these learned rules, thereby influencing the ranking adjustments of unrelated bugs.When the proportion of bugs relative to the total number of alarms is excessively high, false generalization may occur. This issue primarily arises because many bugs share common path nodes, especially in the case of taint analysis, while the LLM can only learn from a subset of alarm paths at a time. As a result, the weights of bug paths that are not learned diminish accordingly, increasing the likelihood of false generalization. To mitigate this issue, further classification and clustering of alarms and bugs may be required. Despite the more pronounced false generalization observed for BINLLM on sdop-0.61, its overall efficiency remains superior to both BINGO and even BAYESMITH, owing to the effectiveness of its rule-based approach.

Additionally, Fig. 6 highlights two key aspects of BINLLM's improvement in error detection efficiency: an increase in the initial rank of true alarms and a faster rise in true alarm ranks. For example, in tar-1.28, the initial ranking for true alarms improved from 697 with BINGO to 112 with BINLLM. This, coupled with a steeper slope for true alarm rankings, indicates a more rapid improvement per iteration, as shown in grep-2.19 and patch-2.7.1, where true alarms, despite initially ranking lower than in BAYESMITH, achieved faster rank progression with fewer iterations.

5.4 Real world case study

To further validate the applicability of BINLLM in more advanced and realistic scenarios and to minimize potential bias from LLM training data, we conducted experiments on the libtiff-4.6.0 dataset. Libtiff-4.6.0 was released in September 2023 and is known to contain a null pointer dereference vulnerability (CVE-2024-7006)(MITRE, 2024), which was publicly discussed in November 2023(libtiff developers, 2024) and subsequently fixed in the latest version, libtiff-4.7.0. For libtiff-4.6.0, we tested BINLLM using two different LLM models, namely BINLLM-4o and BINLLM-gpt4, corresponding to the previously used gpt-4o and the newly incorporated gpt-4-32k, respectively. Notably, gpt-4o has a knowledge cutoff of October 2023, whereas gpt-4-32k has a knowledge cutoff of September 2021. This implies that neither model has been exposed to CVE-2024-7006 during training, and for gpt-4-32k, libtiff-4.6.0 represents an entirely new codebase. The experimental results on CVE-2024-7006 in libtiff-4.6.0 provide an insightful assessment of BINLLM's capability to detect vulnerabilities that are absent from LLM training data and to reduce the need for manual inspection. While this evaluation does not fully capture all the complexities of real-world vulnerability detection, it offers a meaningful approximation of BINLLM's performance in practical security analysis scenarios.

Table 6 Case study of libtiff-4.6.0 with CVE-2024-7006. Effectiveness refers to the number of inspections required to identify a real vulnerability; freq and mag represent the frequency and magnitude of false generalization, respectively.

Method	Effectiveness	Freq	Mag	Freq×Mag
BINGO	101	0	0.0	0.0
BINLLM-4o	55	6	6.7	40.0
BINLLM-gpt4	46	3	10.7	32.0

Table 6 presents the performance metrics of BINLLM-4o, BINLLM-gpt4, and the baseline BINGO on the libtiff-4.6.0 dataset. In terms of check efficiency, BINLLM-4o and BINLLM-gpt4 achieve an average improvement of 50%, meaning that a vulnerability that originally required 101 checks can now be detected in only half that number with the support of rule learning. Although both BINLLM-4o and BINLLM-gpt4 exhibit a slight increase in the frequency and magnitude of false generalizations, these increases have minimal impact on the overall ranking history.

Unedited

5.4.1 Cost

We also evaluated the time and application programming interface (API) costs associated with BinLLM. Our analysis revealed that the primary time bottleneck lies in the extraction of both intra-procedural and inter-procedural paths for all alarms. In this case study, we extracted paths for a total of 510 alarms, with an average extraction time of 0.21h per alarm. In the previously introduced benchmark, despite optimizations for handling recursive calls and excessively long call paths, the path extraction time for a single alarm could still reach 3–4h in complex scenarios. This significantly hinders the scalability of BinLLM when applied to larger codebases. While selective path extraction could be employed in each Bayesian inference iteration based on the alarms requiring further learning, this approach contradicts the real-time nature of BinLLM. Regarding API consumption for LLMs, in this case study, the learning process for a single alarm incurred an average cost of 20.3k prompt tokens and 7.9k completion tokens for gpt-4o, amounting to a total cost of $0.13. For gpt-4-32k, the average consumption was 19.9k prompt tokens and 4.8k completion tokens, with a total cost of $0.89. In this project, by selectively using only a subset of alarms as learning inputs, the API cost per codebase remains low. This efficiency is enabled by BinLLM's staged design, which avoids excessively long contexts while optimizing token consumption.

The case study on libtiff-4.6.0 clearly illustrates the analysis and generalization capabilities of BinLLM in real-world scenarios. Notably, the model's ability to learn about alarms is derived from a detailed analysis and abstraction of the alarm code itself, rather than merely extracting information from an existing database. By leveraging the learning and reasoning capabilities of LLMs, BinLLM further enhances its effectiveness in dynamic environments.

6 Limitations opportunities

This section discusses the limitations of our approach and outlines directions for future research.First, our method still faces significant challenges in terms of time efficiency when applied to large-scale codebases. As previously mentioned, the primary bottleneck lies in the overhead associated with inter-procedural analysis during alarm path extraction. While inter-procedural analysis provides more detailed semantic information about alarms, the fact that a single alarm may require over several hours of processing remains impractical. In BinLLM, we currently use Joern's Code Property Graph (CPG) as the analysis target. Moving forward, we plan to explore lower-level tools like LLVM to accelerate inter-procedural analysis. Additionally, for overly complex inter-procedural call structures, we may consider degrading the analysis to intra-procedural path extraction only, allowing the LLM to determine the appropriate depth for call chain analysis.Furthermore, the effectiveness of BinLLM is not consistently significant across all codebases, and in some cases, an increase in false generalization has been observed. This issue arises from the constraints on the number of learning iterations and the selection of alarms for training. To control computational costs, BinLLM learns only from a subset of high-probability alarms, leaving considerable room for improvement in its generalization capability. Moreover, high-probability alarms often exhibit strong correlations, as they frequently originate from the same function, further limiting the diversity of alarms learned by BinLLM.A key focus of our future work is to develop more effective strategies for selecting representative alarms from alarm reports to enhance learning. We plan to design a more targeted alarm selection method by integrating syntactic and semantic features of alarm paths, ensuring a more diverse and representative learning set for BinLLM.

7 Related works

Our work relates to Bayesian program analysis and static analysis with LLMs. Below, we provide an overview of relevant previous work in these areas:

7.1 Bayesian program analysis

Bayesian program analysis builds probabilistic models based on program features and logical rules, incorporating user feedback to derive posterior information, propagate confidence scores, and compute the likelihood of each alarm. Bingo (Raghothaman et al., 2018) was the first to integrate Bayesian inference with user feedback-driven alarm identification, leveraging posterior knowledge to calculate alarm probabilities. Drake (Heo et al., 2019) ex-

Unedited

plores the relationship between alarms and code changes by comparing different program versions, while DynaBoost (Chen et al., 2021b) uses dynamic analysis to identify data flow behaviors in static analysis. BayeSmith (Kim et al., 2022) refines rules by comparing probabilistic models on training programs to mitigate false generalization, optimizing the Bayesian network by introducing new input tuples. BinGraph (Zhang et al., 2024) adopts a data-driven approach, modifying abstractions to enable learning. Similar to BayeSmith, our approach refines rules by introducing tuples, but unlike BayeSmith, which utilizes training programs to detect false generalization, we extract new tuples directly from the program under analysis by examining syntactic and semantic features of high-probability alarms.

7.2 Static analysis with LLMs

To date, there have been numerous efforts to integrate LLMs with static analysis. (Mohajer et al., 2023) uses LLMs to predict the truth value of warnings by inputting static tool outputs, such as warnings from Infer, along with code snippets. The VulBench dataset (Gao et al., 2023) has been proposed to assess the vulnerability detection capabilities of LLMs, benchmarking them against deep learning models and traditional static analysis tools. Lift(Li et al., 2024a, 2023) detects use-before-initialization (UBI) errors in the Linux kernel by feeding LLMs problems that symbolic execution could not resolve, thereby augmenting static analysis tools. IRIS (Li et al., 2024b) enables LLMs to identify taint information in third-party library APIs, which static analysis then uses for vulnerability detection. In these approaches, LLMs typically function as part of the input or output process within static analysis, rather than serving as an integrated component of the analysis pipeline. In contrast, our approach not only leverages syntactic and semantic features extracted by static analysis but also inputs the abstracted rules into the alarm probability model, achieving a more integrated combination of LLMs and static analysis. This enables us to harness the abstraction and generalization capabilities of LLMs alongside the detection capabilities of static analysis.

8 Conclusion

We introduce BinLLM, an innovative Bayesian alarm learning framework that integrates LLMs with static analysis to optimize alarm processing. BinLLM leverages alarm path and key statement extraction to abstract Datalog rules, enhancing the structure of the alarm probability model. By combining the strengths of static analysis and LLMs, BinLLM improves inference quality within a limited number of learning iterations, significantly reducing false generalization in alarm ranking. Our experimental results demonstrate the effectiveness of BinLLM, achieving improvements of 40.1% and 9.4% over two state-of-the-art baselines in identifying all vulnerabilities within static analysis alarms.This approach illustrates the potential of fusing LLMs with static analysis.

Unedited

Contributors

Xinlong PAN designed and conducted the main experiments, analyzed the data, and drafted the manuscript. Jianhua LI and Zhihong ZHOU supervised the research and provided key guidance throughout the project. Gaolei LI contributed to the model implementation and experimental setup. Xiuzhen CHEN, Jin MA, Jun WU, and Quanhai ZHANG assisted in improving the experimental design and reviewing the manuscript for accuracy and clarity.

Conflict of interest

All the authors declare that they have no conflict of interest.

Data availability

The data that support the findings of this study are available from the authors upon reasonable request.

References

Soufflé: A datalog synthesis tool - rules, (Accessed: 2024-11-04). https://souffle-lang.github.io/rules

Achiam J, Adler S, Agarwal S, et al., 2023. Gpt-4 technical report. *https://arxivorg/abs/230308774*, .

Ahmed T, Pai KS, Devanbu P, et al., 2023. Improving few-shot prompts with relevant static analysis products. *https://arxivorg/abs/230406815v1*, .

Beller M, Bholanath R, McIntosh S, et al., 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. IEEE 23rd Int Conf on Software Analysis, Evolution, and Reengineering, 1:470-481. https://doi.org/10.1109/SANER.2016.105

Chen M, Tworek J, Jun H, et al., 2021a. Evaluating large language models trained on code. *https://arxivorg/abs/210703374*, .

Chen TY, Heo K, Raghothaman M, 2021b. Boosting static analysis accuracy with instrumented test executions. Proc 29th ACM Joint Meeting on European Software Engineering Conference and Symp on the Foundations of Software Engineering, p.1154-1165. https://doi.org/10.1145/3468264.3468626

Christakis M, Bird C, 2016. What developers want and need from program analysis: an empirical study. Proc the 31st IEEE/ACM Int Conf on automated software engineering, p.332-343.

libtiff developers, 2024. Issue #624 - libtiff. https://gitlab.com/libtiff/libtiff/-/issues/624 [Accessed on Feb. 18, 2025].

Eggert P, 2010. sort: Commit 14ad7a2, (Accessed: 2024-11-04). http://git.savannah.gnu.org/cgit/coreutils.git/commit/?id=14ad7a2

Feng ZY, Guo DY, Tang DY, et al., 2020. CodeBERT: A pre-trained model for programming and natural languages. Findings of the Association for Computational Linguistics: EMNLP 2020, Online, p.1536-1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

Ferrante J, Ottenstein KJ, Warren JD, 1987. The program dependence graph and its use in optimization. *ACM Trans on Program Lang and Syst*, 9(3):319-349. https://doi.org/10.1145/24039.24041

Gao ZY, Wang H, Zhou YC, et al., 2023. How far have we gone in vulnerability detection using large language models. *https://arxivorg/abs/231112420*, .

Heo K, Oh H, Yi K, 2017. Machine-learning-guided selectively unsound static analysis. IEEE/ACM 39th Int Conf on Software Engineering, p.519-529. https://doi.org/10.1109/ICSE.2017.54

Heo K, Raghothaman M, Si X, et al., 2019. Continuously reasoning about programs using differential bayesian inference. Proc 40th ACM SIGPLAN Conf on Programming Language Design and Implementation, p.561-575. https://doi.org/10.1145/3314221.3314616

Ji Z, Lee N, Frieske R, et al., 2023. Survey of hallucination in natural language generation. *ACM Comput Surv*, 55(12):248. https://doi.org/10.1145/3571730

Khanfar H, Lisper B, Masud AN, 2015. Static backward program slicing for safety-critical systems. Reliable Software Technologies–Ada-Europe 2015: 20th Ada-Europe Int Conf on Reliable Software Technologies, June 22-26, 2015, Proceedings 20, p.50-65. https://doi.org/10.1007/978-3-319-19584-1_4

Kim H, Raghothaman M, Heo K, 2022. Learning probabilistic models for static analysis alarms. Proc 44th Int Conf on Software Engineering, p.1282-1293. https://doi.org/10.1145/3510003.3510098

Li HN, Hao Y, Zhai YZ, et al., 2023. The hitchhiker's guide to program analysis: A journey with large language models. *https://arxivorg/abs/230800245*, .

Li HN, Hao Y, Zhai YZ, et al., 2024a. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proc ACM Program Lang*, 8(OOPSLA1):111. https://doi.org/10.1145/3649828

Li ZY, Dutta S, Naik M, 2024b. Iris: Llm-assisted static analysis for detecting security vulnerabilities. *https://arxivorg/abs/240517238*, .

Lisper B, Masud AN, Khanfar H, 2015. Static backward demand-driven slicing. Proc Workshop on Partial Evaluation and Program Manipulation, p.115-126. https://doi.org/10.1145/2678015.2682538

Ma W, Liu SQ, Lin ZH, et al., 2023. Lms: Understanding code syntax and semantics for code analysis. *https://arxivorg/abs/230512138*, .

Mangal R, Zhang X, Nori AV, et al., 2015. A user-guided approach to program analysis. Proc 10th Joint Meeting on Foundations of Software Engineering, p.462-473. https://doi.org/10.1145/2786805.2786851

Meyering J, 2018. tar: Commit b531801, (Accessed: 2024-11-04). http://git.savannah.gnu.org/cgit/tar.git/commit/?id=b531801

MITRE, 2015a. Cve-2015-1345, (Accessed: 2024-11-04). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1345

MITRE, 2015b. Cve-2015-8106, (Accessed: 2024-11-04). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8106

MITRE, 2016. Cve-2016-10713, (Accessed: 2024-11-04). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10713

MITRE, 2017a. Cve-2017-16663, (Accessed: 2024-11-04). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE- 2017- 16663

MITRE, 2017b. Cve-2017-9181, (Accessed: 2024-11-04). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE- 2017- 9181

MITRE, 2018. Cve-2018-10372, (Accessed: 2024-11-04). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE- 2018- 10372

MITRE, 2019a. Cve-2019-16166, (Accessed: 2024-11-04). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16166

MITRE, 2019b. Cve-2019-18397, (Accessed: 2024-11-04). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-18397

MITRE, 2024. Cve-2024-7006, (Accessed: 2024-11-04). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-7006

Mohajer MM, Aleithan R, Harzevili NS, et al., 2023. Skipanalyzer: An embodied agent for code analysis with large language models. *https://arxivorg/abs/231018532*, .

Mooij JM, 2010. libDAI: A free and open source C++ library for discrete approximate inference in graphical models. *J Mach Learn Res*, 11:2169-2173.

Muske T, Serebrenik A, 2022. Survey of approaches for postprocessing of static analysis alarms. *ACM Comput Surv*, 55(3):48. https://doi.org/10.1145/3494521

Oh H, Heo K, Lee W, et al., 2012. The sparrow static analyzer, (Accessed: 2024-11-04). https://github.com/ropas/sparrow

Pearce H, Tan B, Ahmad B, et al., 2023. Examining zero-shot vulnerability repair with large language models. IEEE Symposium on Security and Privacy, p.2339-2356. https://doi.org/10.1109/SP46215.2023.10179324

Unedited

Pei KX, Bieber D, Shi KS, et al., 2023. Can large language models reason about program invariants? Proc 40th Int Conf on Machine Learning, p.27496-27520.

Raghothaman M, Kulkarni S, Heo K, et al., 2018. User-guided program reasoning using bayesian inference. Proc 39th ACM SIGPLAN Conf on Programming Language Design and Implementation, p.722-735. https://doi.org/10.1145/3192366.3192417

Rühsen T, 2018a. wget: Commit b3ff8ce, (Accessed: 2024-11-04). http://git.savannah.gnu.org/cgit/wget.git/commit/?id=b3ff8ce

Rühsen T, 2018b. wget: Commit f0d715b, (Accessed: 2024-11-04). http://git.savannah.gnu.org/cgit/wget.git/commit/?id=f0d715b

Shen HH, Fang JH, Zhao JJ, 2011. Efindbugs: Effective error ranking for findbugs. 4th IEEE Int Conf on software testing, verification and validation, p.299-308. https://doi.org/10.1109/ICST.2011.51

Sun YQ, Wu DY, Xue Y, et al., 2024. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. *Proc IEEE/ACM 46th Int Conf on Software Engineering*, article 166. https://doi.org/10.1145/3597503.3639117

Touvron H, Martin L, Stone K, et al., 2023. Llama 2: Open foundation and fine-tuned chat models. *https://arxivorg/abs/230709288*, .

Wei J, Wang X, Schuurmans D, et al., 2022. Chain-of-thought prompting elicits reasoning in large language models. *Proc 36th Int Conf on neural information processing systems*, article 1800.

Yao SY, Yu D, Zhao J, et al., 2023. Tree of thoughts: Deliberate problem solving with large language models. *Proc 37th Int Conf on Neural Information Processing Systems*, article 517.

Zhang YF, Shi YF, Zhang X, 2024. Learning abstraction selection for bayesian program analysis. *Proc ACM Program Lang*, 8(OOPSLA1):128. https://doi.org/10.1145/3649845

Zhou X, Cao SC, Sun XB, et al., 2024. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Trans Softw Eng Methodol, in press*, . https://doi.org/10.1145/3708522

Unedited