Agents as Neuro-Symbolic Reasoners: Path Feasibility Reasoning for Precise Static Bug Detection

Xueying Du[1] **Kai Yu**[1] **Chong Wang**[2] **Yi Zou**[1]
Wentai Deng[1] **Zuoyu Ou**[1] **Xin Peng**[1*] **Yiling Lou**[1]

[1]Fudan University [2]Nanyang Technological University

{xueyingdu21,23262010042,zouy25,wtdeng24}@m.fudan.edu.cn
{22210240250,pengxin,yilinglou}@fudan.edu.cn
chong.wang@ntu.edu.sg

Abstract

Static bug analyzers play a crucial role in ensuring software quality. However, existing analyzers for bug detection in large-scale codebases often suffer from high false positive rates. This limitation largely stems from their inadequate capabilities in performing precise path feasibility validation for complex code contexts. While recent work has explored Large Language Models (LLMs) to eliminate false positives in static bug detection, the limited reasoning capabilities of LLMs over long and complex contexts hinder their effectiveness when applied to large-scale software projects. To address this challenge, we propose PATHAGENT, an agent-driven framework for fine-grained path feasibility analysis. PATHAGENT decomposes complex inter-procedural analysis into localized symbolic range reasoning sub-tasks, and employs an on-demand contextual exploration strategy to adaptively retrieve only the necessary code context. These designs enable precise path feasibility reasoning and effectively reduce false positives reported by static bug analyzers. Evaluation on large-scale real-world projects shows that PATHAGENT eliminates 79% of false positives, outperforming all baselines by 23% to 44%, while maintaining strong bug detection capability with a recall of 0.94.

1 Introduction

Static analysis tools (SATs) play a crucial role in ensuring software quality. They help identify potential bugs early in the development process by scanning the codebase without extensive program execution [19, 21, 20, 35, 36, 37, 39, 42]. Popular analyzers like GitHub CodeQL [7] and CppCheck [2] are widely used for bug detection. Their analysis of many bug types (such as Null-Pointer-Dereference and Use-After-Free) typically follows a standardized pattern: first identifying sources (i.e., potential origins of issues) and sinks (i.e., locations where issues manifest) based on predefined rules, and then checking whether a feasible execution path exists from sources to sinks. If such a path is found, the tools report a potential bug.

However, when applied to complex software projects, SATs often suffer from high false positive rates due to their limited capabilities in accurately performing path feasibility analysis. For example, our manual inspection of sampled bug reports reveals that state-of-the-art tools (e.g., CodeQL and Infer) exhibit extremely high false positive rates (e.g., over 90%) of false positives for Null-Pointer-Dereference (NPD) bugs in large-scale projects such as the Linux Kernel. This problem primarily arises from two key limitations: (1) *Path analysis limitations*: the presence of deeply nested branches, complex computations, and sophisticated control-flow structures makes it chal-

*Corresponding author

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

lenging for SATs to accurately determine which program branch will be executed at runtime. (2) *Contextual analysis limitations:* path feasibility analysis from source to sink often involves long, intricate function call chains with complex data dependencies. Such complexity makes it difficult for SATs to analyze each function return values and perform precise program behavior reasoning. As a result, the large volume of false positives generated by SATs significantly burdens developers, as substantial manual effort is required to review and filter out these false alarms.

Existing Techniques. While model checking and symbolic execution can perform path feasibility analysis to reduce false positives, they suffer from path explosion when traversing all possible paths, particularly when dealing with multiple conditional branches, loops, nested function invocations, or compound data types [18]. Therefore, they fail to scale to large and complex software projects in practice. Given the promising capabilities of Large Language Models (LLMs) in code comprehension [32, 33, 22, 23], recent work leveraged LLMs to reduce false positives reported by static analyzers. Although showing promising effectiveness in synthetic and small-scale programs (e.g., Juliet [10]), these techniques have limitations when applied to large codebases as they leverage LLMs for code analysis in a coarse-grained way. For example, LLM4SA [41] queries LLMs by providing all the code contexts (e.g., callers and callees of the sink, including hundreds of lines of code) into one single prompt. For LLMDFA [40], it prompts LLMs to identify the path feasibility within each function by providing the entire function body in one single prompt. However, for complex projects such as the Linux kernel, there can be hundreds of lines of code and 50+ if-conditions even in one function. As LLMs suffer from limited complex reasoning capabilities, especially over long contexts [30], these techniques have limited effectiveness when applied to large-scale software projects.

Our Approach. To address these limitations, we propose an LLM agent-driven path feasibility analysis framework, PATHAGENT, designed to reduce false positives in static bug detection for large software projects. Specifically, given a function call trace from the source to the sink, PATHAGENT conducts a two-phase, function-by-function analysis: (1) *Path Constraints Extraction*, leveraging an LLM agent to perform symbolic range reasoning over variables and function calls along the key conditional branches of the path, progressively extracting path constraints; and (2) *Constraint Solving*, formalizing the extracted constraints into SMT scripts and leveraging a constraint solver (e.g., Z3) to determine path feasibility. Overall, PATHAGENT introduces the following key innovations compared to traditional symbolic execution and existing LLM-based approaches:

- *Fine-grained decomposition for path analysis.* PATHAGENT decomposes inter-procedural path feasibility analysis into a set of localized symbolic range reasoning sub-tasks that are more tractable for LLMs, thereby reducing reasoning complexity and improving analysis reliability.
- *Agent-driven on-demand contextual exploration.* PATHAGENT incorporates an agent-guided adaptive planning mechanism to selectively explore only the necessary code context, effectively mitigating path explosion in deeply nested inter-procedural call chains.

Evaluation. To enable a more realistic evaluation, we first construct a new dataset, SAFP-BENCH-C, with approximately 150 person-hours of manual effort. The dataset comprises 443 bug warnings reported by three state-of-the-art static analyzers (i.e., CodeQL [7], CppCheck [2], and Infer [9]) on three large-scale real-world software projects, including the Linux Kernel (over 18 million lines of code), OpenSSL (over 300K lines), and Libav (over 600K lines). The dataset includes five critical bug types, i.e., Null-Pointer-Dereference (NPD), Use-After-Free (UAF), Buffer Overflow (BOF), Use-Before-Initialization (UBI), and Divide-by-Zero (DBZ). Our evaluation shows that PATHAGENT successfully filters out 79% false positives reported during static bug detection, substantially outperforming all the baselines by 23% to 44% improvements; meanwhile PATHAGENT maintains strong bug detection capability with a recall of 0.94. Moreover, PATHAGENT shows consistent effectiveness on larger codebases with multiple branch conditions, demonstrating its advantages in enhancing complex inter-procedural path feasibility analysis. PATHAGENT shows consistent effectiveness across four state-of-the-art LLMs, and our ablation studies confirm the contribution of each key component proposed in PATHAGENT. This paper makes the following contributions:

- **A novel technique**: We propose PATHAGENT, an LLM agent-driven path feasibility analysis framework to reduce false positives in static bug detection. PATHAGENT decomposes inter-procedural analysis into localized symbolic range reasoning sub-tasks with agent-driven adaptive contextual exploration, enabling more precise path analysis and on-demand contextual reasoning.
- **A new dataset**: We construct SAFP-BENCH-C, a dataset for evaluating false positive reduction techniques on real-world large-scale software projects.

- **An extensive evaluation**: We conduct comprehensive experiments demonstrating that PATHAGENT effectively reduces false positives while maintaining strong bug detection capability. *Data and code of our work are at [16].*

2 Preliminaries

Source and Sink Definition. Many static bug detection tasks (e.g., Null Pointer Dereference) follow a common pattern: a *target variable* originating from the *source* (where a potential issue originates) reaches the *sink* (where the issue manifests) through a feasible execution path.

Definition (Sources): The program points where a potential bug condition originates. Formally, for a target variable x, a source is the program point immediately after executing a statement such that x satisfies a bug-relevant post-condition (i.e., the target variable x reaches a certain condition value). For instance, in Null Pointer Dereference (NPD) detection, a source may be a program point where x is explicitly assigned `NULL`:

$$\rho_0[e(x)] \Downarrow v_c$$

where $e(x)$ is the evaluation expression for x, and v_c is the certain condition value (e.g., `NULL` for NPD). The symbol $\Downarrow$ denotes the evaluation relation, indicating that expression $e(x)$ evaluates to v_c under state ρ_0.

Definition (Sinks): The program points where the bug potentially manifests, typically where a critical operation is performed on the target variable. For NPD, this would be any dereference of x. More generally, a sink corresponds to a predefined target operation $op \in O$ that is applied to x. Static analysis tools, such as CodeQL and Infer, identify sources and sinks through built-in rule sets (e.g., semantic patterns) based on the bug type.

Path Feasibility Analysis Definition. The core challenge in static bug detection lies in determining whether a feasible execution path exists from a source to a sink, with the target variable's state consistently satisfying a certain condition value along the path. Given a source state ρ_0 and a sink condition involving the target variable x, path feasibility is defined as follows:

Definition (Path Feasibility): There exists a finite execution trace:

$$\rho_0 \to \rho_1 \to \cdots \to \rho_k$$

such that:

- **Source condition**: $\rho_0[e(x)] \Downarrow v_c$ (i.e., x satisfies the target condition at the source post-state),
- **Sink condition**: The target operation $op \in O$ is applied to x at ρ_k,
- **State preservation**: $\rho_i[e(x)] \Downarrow v_c$ for all $i \in [0, k]$ (i.e., x remains in the target condition across all intermediate states).

where ρ_0 is the post-state of the source program point, and ρ_k is the post-state of the sink point.

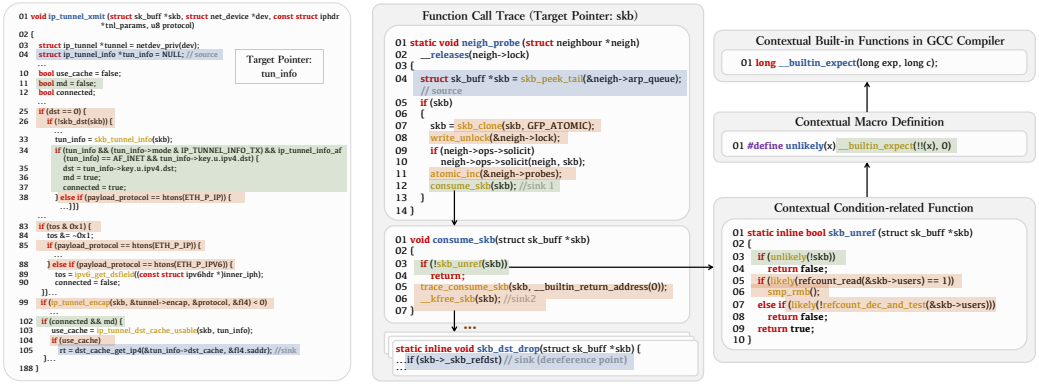
Path Feasibility Analysis can be formalized as the below procedure, which returns *True* if any feasible execution path exists between *source* and *sink*, and *False* otherwise.

$$ExecutionTrace : \{\rho_0 \to \rho_1 \to \cdots \to \rho_k\} \to \{True, False\}$$

3 Motivating Examples

To illustrate the key challenges of path feasibility analysis and motivate our design, we present two real-world Null-Pointer-Dereference false positives reported by CodeQL on Linux kernel v6.9.6. Existing LLM-based techniques (i.e., LLM4SA and LLMDFA) all fail to identify these false positives due to the complexity in path and contextual analysis.

Challenge-1: Complex Constraint Cascades in Path Analysis. Figure 1(a) shows a false positive arising from CodeQL's inability to reason over cascading path constraints. As highlighted in green, The dereference of the target pointer *tun_info* at line 105 (the sink) can only execute if both *connected* and *md* are *TRUE* (line 102). However, *md* is set to *TRUE* (line 36) exclusively when *tun_info* is non-null (line 34), thereby rendering the null pointer dereference infeasible. Correctly



(a) Complex Constraint Cascades (b) Complex Constraint-Related Contextual Analysis

Figure 1: Motivating Examples

identifying this requires jointly reasoning over multiple cascading conditions (highlighted in green and orange) distributed across the function. Existing LLM-based techniques that feed entire function bodies into a single prompt overwhelm LLMs, making it difficult to focus on critical constraints and accurately determine path feasibility.

Challenge-2: Deep Contextual Analysis. Figure 1(b) illustrates a false positive caused by insufficient contextual reasoning. In *consume_skb*, the null check on the target pointer *skb* is performed via the callee *skb_unref* (line 3), which triggers an early return when *skb* is null, preventing it from reaching the sink. Determining this requires reasoning across multiple layers of function calls, macro expansions, and GCC built-in semantics. Moreover, much of the relevant context lies outside the direct source-to-sink trace and is embedded in constraint-related callees. Notably, Figure 1(b) presents only the minimal context (highlighted in green) required for manual inspection. In contrast, existing approaches typically include a much broader set of contextual callees (highlighted in orange), significantly increasing reasoning complexity and posing challenges for both LLM-based and traditional symbolic execution techniques.

4 Approach

To address the challenge of complex reasoning with long context, *our key insight is to decompose complex and ambiguous reasoning into fine-grained sub-tasks with explicit objectives that are more tractable for LLMs.* Figure 2 shows the overview of our approach. Given a target variable, and the function call trace from source to sink, PATHAGENT performs path feasibility analysis function-by-function. Each function is analyzed by collecting feasible path constraints and solving them to determine whether a feasible source-to-sink path exists. The process returns *True* if any feasible path exists, and *False* otherwise. The overall trace is deemed path feasible only if all functions along the trace are feasible. The detailed analysis within each function consists of two steps:

- **Feasibility Path Constraints Extraction (Section 4.1)**: PATHAGENT first extracts feasible path conditional expressions, then incorporates LLM agents to conduct symbolic range reasoning for variables and function calls within each expression individually.
- **Constraints Solving (Section 4.2)**: PATHAGENT formalizes the extracted path constraints into SMT query scripts, and leverages the constraint solver (i.e., Z3) to determine path feasibility.

Insight. We decompose the complex path feasibility analysis into localized symbolic range reasoning sub-tasks, reducing the reasoning complexity and improving LLM analysis reliability (*addressing challenge-1*). During inter-procedural reasoning, we introduce an agent-driven on-demand contextual exploration strategy, selectively exploring only the necessary code context, effectively mitigating path explosion in deep contextual analysis (*addressing challenge-2*).

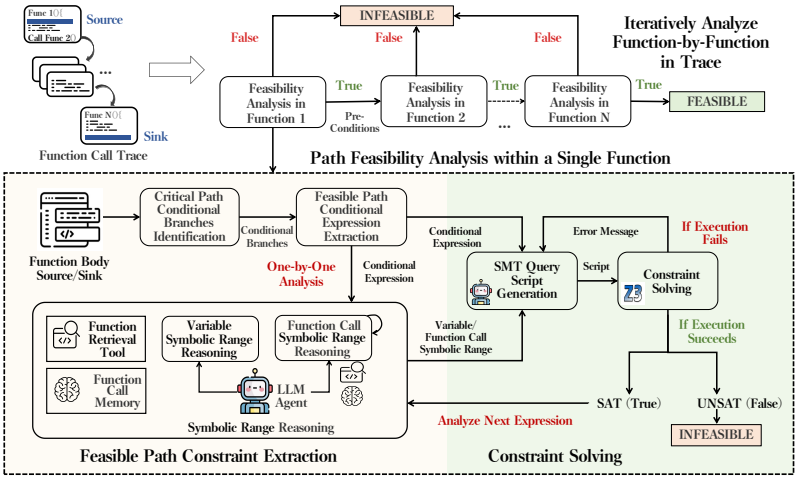


Figure 2: Overview of PATHAGENT

4.1 Feasibility Constraints Extraction

Given a function f with the CFG entry point S_f and exit point T_f, PATHAGENT first extracts the feasible path constraints from S_f to T_f. To improve precision and avoid exhaustive path exploration, we first identify critical path conditional expressions that directly affect the reachability of T_f (Section 4.1.1). We then employ LLM agents to infer symbolic ranges for variables and function return values within these expressions (Section 4.1.2). These conditional expressions and symbolic ranges collectively constitute the feasible path constraints.

4.1.1 Critical Path Conditional Expression Extraction

Based on the extracted CFG of f, PATHAGENT first identifies the critical path conditional branches, which are a set of conditional statement nodes (i.e., branching and loop constructs) that directly impact the reachability of T_f at the control flow level. These branches are represented as:

$$N_{cs}(S_f, T_f) = N_{\text{e}}(T_f) \cup N_{\text{jump}}(S_f, T_f)$$

where $N_{\text{e}}(T_f)$ represents all conditional statement nodes nested outside the basic block containing T_f; and $N_{\text{jump}}(S_f, T_f)$ refers to conditional statement nodes nested outside any basic block containing *Jump/Return* statements encountered between S_f and T_f. For branching statements, we focus on *If* statements (i.e., *if*, *else*, *else if* and *switch*); for *Loop* statements, we focus on *while*, *do* and *for* statements. Regarding *Jump/Return* statements, we consider all *goto* and *return* statements, as well as *break* and *continue* statements within loop constructs that are nested outside T_f.

Based on these critical path conditional branches, PATHAGENT further extracts the feasible path conditional expressions, which are represented as:

$$FPE(S_f, T_f) = \{expr_f \mid expr_f \in N_{\text{e}}(T_f)\}$$
$$\cup \{\neg expr_{if} \mid expr_{if} \in N_{\text{jump}}(S_f, T_f)\}$$

Where a feasible path exists only when all $expr_f$ are satisfied (i.e., the conditions for reaching the exit point branch are fulfilled), and none of $expr_{if}$ is fully satisfied (i.e., there is no early jump to other branches).

4.1.2 Context-Aware Symbolic Range Reasoning

In this step, PATHAGENT utilizes LLM agents to reason all symbolic ranges Σ_v of each feasible path conditional expression one by one. Notably, Σ_v encompasses the symbolic ranges from both variables and the function call return values in conditional expressions. Since these constraint-related function calls lie outside the input call trace, they require additional contextual analysis.

Variable Symbolic Range Reasoning. For each variable v in a conditional expression, PATHAGENT employs LLM agents to simulate execution and infer the possible symbolic ranges $\Sigma_v(P)$

of v when the conditional statement is executed. Here, P denotes the required initial states of the program. The initial states P denotes the post-condition after executing S_f, derived from both its pre-conditions and execution semantics. For the first function in the trace, P is directly defined by the post-condition of S_f. For subsequent functions, P further incorporates all feasible path constraints associated with parameters propagated from the preceding function. This design ensures consistent constraint propagation along the call trace and enables rigorous inter-procedural analysis. The definition of P is bug-type specific. For NPD detection, the *NULL* state of the target pointer is propagated as P throughout the entire trace analysis.

Inspired by chain-of-thought (CoT), PATHAGENT directs the LLM agent to perform a step-by-step simulated execution of function code, enabling more precise reasoning by the LLM. Furthermore, example analyses are included within the prompt to facilitate few-shot learning for LLM agents. The detailed prompt design is in Appendix C

Function call symbolic range reasoning with on-demand contextual exploration. For constraint-related function calls fc within a conditional expression, PATHAGENT aims to determine the symbolic range $\Sigma_f(P)$ of the function return value under the required initial states P. The definitions of P align with $\Sigma_v(P)$. Figure 3 illustrates the framework for reasoning symbolic ranges of constraint-related function call return values with on-demand contextual exploration. To address the challenges posed by the deep multi-layered contextual analysis (Challenge-2), we propose an agent-driven framework that leverages the tool-usage and self-planning capabilities of LLM agents for on-demand, precise contextual analysis.

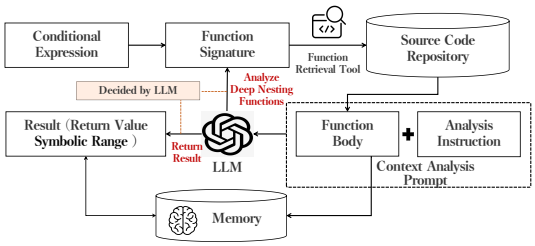


Figure 3: Agent-driven function call symbolic range reasoning with on-demand exploration

As feasible path conditional expressions may involve multiple function calls, only those with parameters relevant to P are considered for symbolic range reasoning. For each, PATHAGENT retrieves its function body from the source code repository using a function retrieval tool. This function body, along with the initial states P, is then provided to LLMs for reasoning. During the reasoning process, the LLM agent leverages its self-planning capabilities to adaptively determine whether deeper contextual functions are required to refine the analysis. Contextual functions identified as affecting the return values are recursively analyzed with the same process. This process continues until the LLM agent finalizes the result or reaches the predefined maximum iteration limit (5 in this work). The complete procedure for recursive contextual exploration in symbolic range reasoning is described in Algorithm 1 in the Appendix B. The detailed prompt design is in Appendix C

To minimize resource consumption due to redundant analyses of identical function calls, we introduce a memory module. This module stores the return value symbolic range of previously analyzed function calls under distinct P. In subsequent analysis, results for identical scenarios can be retrieved directly from the memory module.

4.2 Constraints Solving

Given the feasible path constraints for T_f, PATHAGENT iteratively employs LLM to convert them into SMT query scripts, and invokes Z3 solver for constraint solving.

When analyzing complex functions, PATHAGENT may solve numerous feasible path constraints. Directly converting all constraints into SMT query scripts at once can overwhelm the LLM. To address this, we propose an iterative generation strategy, consisting of three key steps: (1) **Script template generation.** We utilize LLM to generate a Python template of the SMT query script, including imported packages and constraint solver definitions; (2) **SMT constraints generation.** With few-shot examples, LLMs then convert path constraints into SMT constraints (i.e., *z3.add()* format). (3) **Script merging.** Lastly, LLMs merge SMT constraints with the template to create the final script, ensuring consistency between SMT variable definitions and SMT constraints. After generating the initial script, the first round of constraint solving is performed. If the solver returns an

UNSAT (i.e., infeasible) result, further symbolic range reasoning is terminated. During this process, each set of symbolic ranges in conditional expressions inferred by the LLM will be converted into SMT constraints and immediately merged into the script for constraint solving. Once the solver returns *UNSAT*, further reasoning is halted to avoid inefficient analysis of additional constraints.

To address script generation errors, PATHAGENT incorporates error messages from the Z3 solver into the prompt, enabling targeted fixes by the LLM. This process repeats until the Z3 solver successfully resolves the constraints or the maximum number of fix attempts (3 rounds in our setting) is reached. The detailed prompt design of constraint solving is in Appendix C.

5 Evaluation

Dataset. Previous false alarm datasets [40, 41] either focus on synthetic small program or not release the complete code contexts. Existing vulnerability detection datasets [24, 43] only includes true alarms and and lack the false alarms that can be identified by SATs, thus being unsuitable for the evaltion. Therefore, we first manually label and construct a new benchmark SAFP-BENCH-C, which includes false positives and real bugs reported by state-of-the-art static analyzers on real-world large-scale software systems.

Bug Types. Following previous work on static bug analysis [41, 40], we target five representative categories of critical bugs: Null-Pointer-Dereference (NPD), Use-after-Free (UAF), Buffer-Overflow (BOF), Divided-by-Zero (DBZ), and Use-before-Initialization (UBI). **Target Projects.** We select the recent versions of three large-scale and well-maintained C/C++ open-source projects for scanning, including the Linux kernel (v6.9.6) [12], OpenSSL (v3.4.0) [15], and Libav (v12.3) [11]. **Static Analyzers.** We use three state-of-the-art and widely-adopted static analyzers: CodeQL [7], Infer [9], and CppCheck [2]. **Labeling process.** We invite 3 participants to label the warnings manually, with a substantial agreement (i.e., Cohens Kappa of 0.76). In total, the construction process takes around 150 person-hours. **Data statistics.** The final benchmark SAFP-BENCH-C contains 443 warnings, of which 47 are real bugs and 396 are false positives. The detailed benchmark statistics and labeling process are described in Appendix A.

Setup and Metrics. We use three commonly-adopted metrics in vulnerability detection tasks: accuracy, precision, and recall. Additionally, we introduce two metrics to evaluate the effectiveness in identifying false positives generated by static analyzers: False Positive Reduction Precision (FPR_P) and False Positive Reduction Recall (FPR_R), which specifically measure the precision and recall for negative samples (i.e., false positives) in the dataset.

$$\text{FPR_P} = \frac{\text{TN}}{\text{TN} + \text{FN}} \quad \text{FPR_R} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

For both PATHAGENT and the baselines, we use GPT-4o-mini [4] as the backbone LLM. The detailed setup is described in Appendix B.

5.1 RQ1: Method Comparison

Baselines. We mainly compare PATHAGENT against four state-of-the-art LLM-based false positive mitigation methods as follows:

- **LLM4SA** [41] utilizes LLMs to process bug warning reports from various static analysis tools for false positives reduction. For LLM4SA, we use its public implementation.
- **LLMDFA** [40] is a LLM-based dataflow analysis framework. We use LLMDFA for analyzing the reachability between the source and sink to determine false positives.
- **IRIS** [31] leverages LLMs to identify sources and sinks, relies on CodeQL for path analysis, and employs LLM-based false positive filtering. We evaluate its false positive filtering component on our benchmark.
- **RepoAudit** [26] performs repository-level vulnerability detection through LLM-based dataflow analysis. We evaluate its PathValidator component for path feasibility analysis.

We do not include traditional analysis-based techniques (e.g., symbolic execution and model checking) [38, 29, 25], as they suffer from scalability and generalization issues when applied to large-scale systems like the Linux kernel. Furthermore, we do not include supervised learning-based techniques [28, 27] as they require massive labeled training data.

Overall Performance. Table 1 compares PATHAGENT with four baselines on SAFP-BENCH-C. Table 7 and Table 8 in Appendix D respectively compares PATHAGENT with five bug types and three static analyzers. Overall, PATHAGENT significantly outperforms all baselines across all metrics. Furthermore, PATHAGENT eliminates more false positives than the baselines (with an FPR_R improvement of 23% - 44%, $p < 0.01$). Particularly, PATHAGENT precisely filters out 79% false alarms on all five bug types. Meanwhile, PATHAGENT still detected 44 real bugs (true positives) out of 47, achieving a high recall rate of 0.94 (with a 3% - 30% improvement).

Although our approach achieves a "low" overall precision of 0.34, it still substantially outperforms conventional static analyzers, whose precision ranges from 0.05 to 0.12 (i.e., CodeQL: 0.12, CppCheck: 0.11, Infer: 0.05). The relatively low precision observed across all methods is primarily attributed to the inherent complexity of real-world large-scale software systems. For instance, while CodeQL achieves 93.8% precision on small-scale synthetic datasets such as Juliet [10], its precision drops to 12% on our benchmark. This highlights the necessity of benchmarking false positives of static analyzers on complex systems. The detailed analysis of the comparison among bug types and static analyzers is described in Appendix D.

Table 1: Effectiveness of PATHAGENT

Tech.	Acc.	FPR_P	FPR_R	Pre.	Recall
PATHAGENT	**0.80**	**0.99**	**0.79**	**0.34**	**0.94**
LLM4SA	0.60	0.98	0.56	0.20	0.91
RepoAudit	0.54	0.92	0.53	0.14	0.64
IRIS	0.40	0.95	0.34	0.13	0.85
LLMDFA	0.38	0.90	0.35	0.12	0.71

Impact on Code Sizes. To further evaluate the effectiveness of PATHAGENT on large-scale complex code repositories with multiple conditional branches, Table 2 and Table 3 respectively compare the effectiveness of PATHAGENT and two baselines on increasing code sizes and branch conditions. As shown in the two tables, we observe a stable performance of PATHAGENT on larger code sizes, but the two baselines exhibit poorer performance with increasing code sizes and branch conditions. This further demonstrates the significant advantages of PATHAGENT in enhancing complex inter-procedural feasibility analysis.

Table 2: Comparison among Code Length

Lines	Tech.	Acc.	FPR_P	FPR_R	Pre.	Recall
Lines $<$ 100	PathAgent	**0.92**	**1.00**	**0.88**	**0.77**	**1.00**
	LLM4SA	0.64	0.75	0.71	0.44	0.50
	LLM4DFA	0.50	0.90	0.35	0.35	0.90
100 $\leq$ Lines $<$ 200	Path	**0.86**	**0.93**	**0.87**	**0.71**	**0.83**
	LLM4SA	0.60	0.57	0.80	0.67	0.40
	LLM4DFA	0.62	0.89	0.53	0.42	0.83
Lines $\geq$ 200	PathAgent	**0.92**	**0.88**	**1.00**	**1.00**	**0.83**
	LLM4SA	0.38	0.40	0.50	0.33	0.25
	LLM4DFA	0.15	0.17	0.14	0.14	0.17

Table 3: Comparison among Condition Count

Cond. Count	Tech.	Acc.	FPR_P	FPR_R	Pre.	Recall
Cond $<$ 20	PathAgent	**0.93**	**1.00**	**0.90**	**0.77**	**1.00**
	LLM4SA	0.68	0.80	0.71	0.50	0.63
	LLM4DFA	0.55	1.00	0.40	0.35	1.00
20 $\leq$ Cond $<$ 40	PathAgent	**0.84**	**0.92**	**0.85**	**0.71**	**0.83**
	LLM4SA	0.50	0.50	0.80	0.50	0.20
	LLM4DFA	0.47	0.71	0.38	0.33	0.67
Cond $\geq$ 40	PathAgent	**0.91**	**0.83**	**1.00**	**1.00**	**0.83**
	LLM4SA	0.38	0.40	0.50	0.33	0.25
	LLM4DFA	0.18	0.17	0.20	0.20	0.17

5.2 RQ2: Model Versatility

To evaluate the generality of PATHAGENT across different backbone LLMs, we implement PATHAGENT using four state-of-the-art models that are widely adopted in bug detection tasks. These include two closed-source models, i.e., GPT-4o-mini [4], Claude Sonnet 3.5 [1], and two open-source models, i.e., Qwen2.5-Coder-32B-Instruct [5], DeepSeek-V2-Instruct [3]. Table 4 presents the effectiveness of our approach PATHAGENT across four different LLMs. Overall, PATHAGENT demonstrates significant effectiveness on all models (all outperforming baselines using GPT-4o-mini), showcasing the generalizability of our approach across different LLMs.

5.3 RQ3: Ablation Studies and Reliability Analysis

Ablation Settings. We evaluate the effectiveness of our iterative refinement-based technical design by comparing PATHAGENT with three ablation variants: (i) **NOCONTEXT** performs symbolic range reasoning without iterative context analysis driven by the LLM agent; (ii) **BATCHSOL** converting all feasible path constraints into a single SMT query script at once for direct constraint solving, without intermediate iterations; and (iii) **NOSYMREASON** omitting the symbolic range reasoning process, it directly performs constraint solving using the extracted critical path conditional expressions.

Effectiveness of Key Components. Table 5 presents the comparison results of ablation variants with gpt-4o-mini. Overall, PATHAGENT demonstrates a significant superiority over these three ablations in false positive elimination, showing 0.41 - 0.57 increase in FPR_R. While BATCHSOL achieves a recall of 0.95, compared to PATHAGENT's 0.90 in Linux kernel vulnerability detection, this discrepancy arises from a real bug being misclassified as a false alarm by PATHAGENT due to inaccurate extraction of critical path conditional expressions. This case is still considered a bug alarm in BATCHSOL due to a constraint solving failure. A high rate of constraint solving failures in BATCHSOL contributes to its low FPR_R (i.e, 0.38, 57.8% less than PATHAGENT). This highlights the key contribution of LLM-based self-planning context analysis and iterative constraint solving in PATHAGENT.

Reliability of LLM-Based Reasoning. To further assess the reliability of LLM-based symbolic range reasoning and self-planning, we manually examined 50 randomly sampled cases analyzed by PATHAGENT with GPT-4o-mini, including 38 correctly classified and 12 misclassified cases. Among the 12 misclassified cases, eight of the ten false positives resulted from over-approximation in LLM reasoning, including three cases where the agent failed to retrieve necessary contextual functions. Overall, the LLM achieved a correct reasoning rate of 70% (35/50) and successfully retrieved the necessary contextual functions in 94% (47/50). These results demonstrate that PATHAGENT is relatively reliable in practice, with its design reducing the risk of missing real bugs due to under-approximation.

Table 4: Generality with Different LLMs

Type	Model	Acc.	FPR_P	FPR_R	Pre.	Recall
NPD	GPT-4o-mini	**0.75**	**0.98**	**0.72**	**0.36**	**0.93**
	Claude	0.71	**0.98**	0.68	0.32	0.90
	Qwen	0.65	0.96	0.62	0.27	0.83
	Deepseek	0.72	0.97	0.69	0.33	0.88
BOF	GPT-4o-mini	0.86	1.0	0.86	/	/
	Claude	0.86	1.0	0.86	/	/
	Qwen	0.86	1.0	0.86	/	/
	Deepseek	0.86	1.0	0.86	/	/
UAF	GPT-4o-mini	0.96	1.0	0.96	/	/
	Claude	0.96	1.0	0.96	/	/
	Qwen	0.96	1.0	0.96	/	/
	Deepseek	0.96	1.0	0.96	/	/
DBZ	GPT-4o-mini	**1.0**	1.0	**1.0**	**1.0**	**1.0**
	Claude	0.94	1.0	0.93	0.67	1.0
	Qwen	0.94	1.0	0.93	0.67	1.0
	Deepseek	**1.0**	1.0	**1.0**	**1.0**	**1.0**
UBI	GPT-4o-mini	**0.90**	1.0	**0.90**	/	/
	Claude	0.86	1.0	0.86	/	/
	Qwen	0.86	1.0	0.86	/	/
	Deepseek	0.89	1.0	0.89	/	/

Table 5: Ablation of PATHAGENT

Project	Tech.	Acc.	FPR_P	FPR_R	Pre.	Recall
Kernel	PATHAGENT	**0.9**	**0.96**	**0.90**	**0.8**	0.90
	NOCONTEXT	0.44	0.8	0.25	0.35	0.86
	BATCHSOL	0.56	0.95	0.38	0.41	**0.95**
	NOSYMREASON	0.37	0.83	0.10	0.33	**0.95**
Openssl	PATHAGENT	**0.8**	0.97	**0.78**	**0.47**	0.89
	NOCONTEXT	0.46	0.89	0.40	0.22	0.78
	BATCHSOL	0.44	0.93	0.34	0.23	0.89
	NOSYMREASON	0.26	**1.0**	0.10	0.20	**1.0**
Libav	PATHAGENT	**0.51**	**1.0**	**0.49**	**0.07**	**1.0**
	NOCONTEXT	0.21	**1.0**	0.18	0.04	**1.0**
	BATCHSOL	0.23	**1.0**	0.20	0.04	**1.0**
	NOSYMREASON	0.25	**1.0**	0.22	0.04	**1.0**
Total	PATHAGENT	**0.75**	**0.97**	**0.71**	**0.42**	0.91
	NOCONTEXT	0.37	0.88	0.26	0.21	0.85
	BATCHSOL	0.42	0.96	0.30	0.23	0.94
	NOSYMREASON	0.29	0.92	0.14	0.16	**0.97**

5.4 RQ4: Costs Analysis

We compare the runtime of PATHAGENT with baseline methods. PATHAGENT requires 28.9 seconds per trace on average, with 6.3 LLM calls, while LLMSA and LLMDFA take 6.8 and 8.6 seconds, respectively. Although PATHAGENT introduces 20 more seconds than baselines, it delivers substantial performance improvements of 23% - 44% over the baselines. Importantly, false positive mitigation is typically performed offline after applying static analyzers and is not time-sensitive. For instance, running CodeQL on the Linux kernel take around 3 hours, making the additional 20 seconds negligible. In contrast, false positives can incur significant manual debugging costs, often requiring hours of developer effort per case. Therefore, accuracy is more important than offline runtime costs for this task. Overall, we consider the additional offline costs of PATHAGENT are acceptable, making PATHAGENT a practical solution for false positive mitigation.

5.5 RQ5: Bad Case Analysis

To understand the limitation of PATHAGENT, we manually analyze a random sample of 50 bad cases, which primarily fall into three categories.

- **LLM reasoning errors (37 instances, 74% cases)**: The main reason for these bad cases is that LLMs generate incorrect approximations during symbolic range reasoning, as LLMs might hallucinate or fall short in understanding program behaviors.
- **Incorrect extraction for critical path conditional expression (11 instances, 22% cases)**: This is attributed to our parser for condition extraction failing on some corner cases (e.g., multiple identical statements to the bug line exist in different locations of the code).

- **Incorrect SMT scripts (2 instances, 4% cases)**: LLM generated SMT scripts exhibit executable errors. Based on our manual inspection, we find these cases are caused by missing or incorrectly converting critical conditions during generation.

To further evaluate the reliability of the SMT script generation process, we performed a manual analysis by checking the semantic correctness of 50 randomly sampled generated scripts (including good cases and bad cases). For these 50 cases, we invited a participant to manually write SMT scripts based on the LLM reasoning results from the earlier steps, and then compared them with the LLM-generated scripts. The comparison results showed that 94% of scripts are semantically equivalent to human-written ones; the remaining three bad cases were mainly caused by missing conditions mentioned in the LLM analysis (two of them were non-critical conditions, so they did not impact the final results).

6 Related Work

Static Analysis-Based False Positive Elimination. Prior work improves false positive elimination through more precise static analysis, mainly via model checking and symbolic execution. *Model checking-based approaches* abstract the program into a finite-state model and use formal verification tools (e.g.,CBMC [6]) to exhaustively explore all possible execution states in order to verify path feasibility [34, 38]. *Symbolic execution-based approaches* symbolically execute the program to collect path constraints and verify path feasibility using SMT solvers [29, 25]. Despite their precision, both lines suffer from path explosion and tight coupling with specific tools or handcrafted rules, hindering scalability and generalization. Therefore, we exclude these symbolic execution and model checking techniques from our baseline comparisons.

Learning-Based False Positive Elimination. Learning-based methods train predictive models that distinguish true bugs from false positives based on features extracted from source code or warning reports. Lee et al.[28] train a CNN to learn lexical patterns from code segments that trigger alarms. Kharkar et al.[27] adopt a Transformer-based model to identify false positive warnings. However, these supervised learning-based approaches require large-scale labeled training data for each bug type, which limits their generalizability across different bug types or in scenarios where training data is unavailable. Recent work leverages LLMs to identify false positives during static bug detection. Wen et al. [41] propose LLM4SA, which provides LLMs with all the code contexts around sink (e.g., callers and callees of the sink and including hundreds of lines of code) into one single prompt. Wang et al. [40] proposes LLMDFA, which leverages few-shot chain-of-thought prompting to guide LLMs to analyze the path feasibility within each function. These techniques utilize LLMs in a coarse-grained way by overwhelming LLMs with long and complex code contexts, reducing their effectiveness when applied to large-scale systems. In contrast, we decompose complex inter-procedural path feasibility analysis into a sequence of localized symbolic range reasoning subtasks, and deploys an LLM agent to perform on-demand contextual analysis, enabling consistent effectiveness on real-world large-scale systems.

7 Conclusion

This paper proposes PATHAGENT, a fine-grained path feasibility analysis framework powered by LLM agents, which effectively enhances complex inter-procedural feasibility analysis for reducing false positives in static bug detection. Our evaluation shows that PATHAGENT substantially outperforms all baselines in both bug detection (with 0.94 recall) and false positive elimination (successfully eliminating 79% false positives, achieving an improvement in FPR_R ranging from 23% to 44%) across different vulnerability types, static analyzers, and backbone LLMs.

8 ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (Grant No. 62332005 and 62372114).

References

[1] Claude sonnet 3.5. https://www.anthropic.com/news/claude-3-family, 2024.

[2] Cppcheck. http://cppcheck.net/, 2024.

[3] Deepseek-coder-v2. https://huggingface.co/deepseek-ai/DeepSeek-Coder-V2-Instruct, 2024.

[4] Gpt-4-turbo. https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo, 2024.

[5] Qwen2.5-coder-32b- 256 instruct. https://huggingface.co/Qwen/Qwen2.5-Coder-32B-Instruct, 2024.

[6] Cbmc. http://www.cprover.org/cbmc/, 2025.

[7] Codeql. https://codeql.github.com/, 2025.

[8] Ctags. https://github.com/universal-ctags/ctags, 2025.

[9] Infer. https://fbinfer.com/, 2025.

[10] Juliet test suite c/c++. https://samate.nist.gov/SARD/test-suites/112, 2025.

[11] Libav. https://github.com/libav/libav/tree/master, 2025.

[12] Linux-kernel 6.9.6. https://elixir.bootlin.com/linux/v6.9.6/source, 2025.

[13] Lizard. https://github.com/terryyin/lizard, 2025.

[14] Openai api. https://openai.com/api/, 2025.

[15] Openssl. https://github.com/openssl/openssl/tree/master, 2025.

[16] Package. https://github.com/LLM4PFA/LLM-Enhanced-Path-Feasibility-Analysis, 2025.

[17] Tree-sitter. https://tree-sitter.github.io/tree-sitter/, 2025.

[18] R. Baldoni, E. Coppa, D. C. Delia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), May 2018.

[19] Y. Cai, P. Yao, C. Ye, and C. Zhang. Place your locks well: understanding and detecting lock misuse bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3727–3744, 2023.

[20] Y. Cai, P. Yao, and C. Zhang. Canary: practical static detection of inter-thread value-flow bugs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1126–1140, 2021.

[21] Y. Cai, C. Ye, Q. Shi, and C. Zhang. Peahen: Fast and precise static deadlock detection via context reduction. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 784–796, 2022.

[22] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[23] X. Du, G. Zheng, K. Wang, Y. Zou, Y. Wang, W. Deng, J. Feng, M. Liu, B. Chen, X. Peng, et al. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. *ACM Transactions on Software Engineering and Methodology*, 35(10):1–25, 2026.

[24] J. Fan, Y. Li, S. Wang, and T. N. Nguyen. A C/C++ code vulnerability dataset with code changes and CVE summaries. In S. Kim, G. Gousios, S. Nadi, and J. Hejderup, editors, *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, pages 508–512. ACM, 2020.

[25] M. R. Gadelha, E. Steffinlongo, L. C. Cordeiro, B. Fischer, and D. Nicole. Smt-based refutation of spurious bug reports in the clang static analyzer. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 11–14. IEEE, 2019.

[26] J. Guo, C. Wang, X. Xu, Z. Su, and X. Zhang. Repoaudit: An autonomous llm-agent for repository-level code auditing. *arXiv preprint arXiv:2501.18160*, 2025.

[27] A. Kharkar, R. Z. Moghaddam, M. Jin, X. Liu, X. Shi, C. Clement, and N. Sundaresan. Learning to reduce false positives in analytic bug detectors. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 13071316, New York, NY, USA, 2022. Association for Computing Machinery.

[28] S. Lee, S. Hong, J. Yi, T. Kim, C.-J. Kim, and S. Yoo. Classifying false positive static checker alarms in continuous integration using convolutional neural networks. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 391–401. IEEE, 2019.

[29] H. Li, T. Kim, M. Bat-Erdene, and H. Lee. Software vulnerability detection using backward trace analysis and symbolic execution. In *2013 International Conference on Availability, Reliability and Security*, pages 446–454. IEEE, 2013.

[30] T. Li, G. Zhang, Q. D. Do, X. Yue, and W. Chen. Long-context llms struggle with long in-context learning. *arXiv preprint arXiv:2404.02060*, 2024.

[31] Z. Li, S. Dutta, and M. Naik. Iris: Llm-assisted static analysis for detecting security vulnerabilities. In *International conference on learning representations*, volume 2025, pages 35735–35758, 2025.

[32] J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024.

[33] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[34] H. Post, C. Sinz, A. Kaiser, and T. Gorges. Reducing false positives by combining abstract interpretation and bounded model checking. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 188–197. IEEE, 2008.

[35] Q. Shi, R. Wu, G. Fan, and C. Zhang. Conquering the extensional scalability problem for value-flow analysis frameworks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 812–823, 2020.

[36] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 693–706, 2018.

[37] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 254–264, 2012.

[38] M. Valdiviezo, C. Cifuentes, and P. Krishnan. A method for scalable and precise bug finding using program analysis and model checking. In *Asian Symposium on Programming Languages and Systems*, pages 196–215. Springer, 2014.

[39] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25:1419–1457, 2020.

[40] C. Wang, W. Zhang, Z. Su, X. Xu, X. Xie, and X. Zhang. Llmdfa: Analyzing dataflow in code with large language models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[41] C. Wen, Y. Cai, B. Zhang, J. Su, Z. Xu, D. Liu, S. Qin, Z. Ming, and T. Cong. Automatically inspecting thousands of static bug warnings with large language model: How far are we? *ACM Transactions on Knowledge Discovery from Data*, 18(7):1–34, 2024.

[42] H. Yan, Y. Sui, S. Chen, and J. Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*, pages 327–337, 2018.

[43] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 10197–10207, 2019.

A Dataset

SAFP-BENCH-C contains 443 warnings generated by three static analyzers scanning three C/C++ projects, of which 47 are real bugs and 396 are false positives.

Labeling process. We apply each static analyzer to each project with its built-in checkers for each bug type. For the reported bug warnings, we then adopt the following process for manual inspection. First, if a static analyzer detects more than 100 warnings in a project, we randomly sample 100 instances to control manual labeling effort; otherwise, all instances would be included. Then, for each warning, two participants will individually label it as a true positive or a false positive; for the inconsistent cases, a third judge will be introduced to solve the conflicts. If the participants cannot reach an agreement after discussion, the warning would be filtered out. The agreement between the two primary annotators achieves a Cohens kappa of 0.76, indicating substantial agreement.

All the participants have over five years of experience in C/C++, and each would receive 200 dollars for the labeling. Each participant is provided with the following instructions: "Please carefully read the warning message and the complete code context, and then determine whether the warning is a true positive or a false positive." The procedure is approved by the Institutional Review Board (IRB) at our institution. In total, the construction process approximately takes around 150 person-hours.

Data statistics. The final benchmark SAFP-BENCH-C contains 443 warnings, of which 47 are real bugs, and 396 are false positives. Such a dominant distribution of false positives over true positives has been widely observed in previous work [41], confirming the prevalence of false positives of static bug analysis in practice. Table 6 presents the detailed statistics of SAFP-BENCH-C. where each entry is reported as "the total number of warnings (number of real bugs)" Notably, empty cells in the table indicate that the corresponding static analyzer did not report any warnings of that bug category for the given project. Although all analyzers are capable of detecting all five bug categories, and we employed their default checkers without any customization (e.g., executing the default UseAfterFree.ql in CodeQL for UAF detection), certain analyzers may fail to identify specific bug types in particular projects.

Table 6: Statistics of SAFP-BENCH-C

Type	Analyzer	Kernel	OpenSSL	Libav	Total
NPD	CodeQL	70 (22)	50 (8)	57 (2)	177 (32)
	CppCheck	83 (8)	5 (0)	10 (2)	98 (10)
	Infer	/	16 (3)	6 (0)	22 (3)
UAF	CodeQL	16 (0)	9 (0)	/	25 (0)
BOF	CodeQL	/	42 (0)	/	42 (0)
DBZ	CppCheck	15(2)	/	1(0)	16 (2)
UBI	CodeQL	/	/	23(0)	23 (0)
	Infer	/	20 (0)	20 (0)	40 (0)

B Implementation

Critical Path Conditional Expression Extraction. We implement this extraction process in three steps: (1) We first parse the source code using *Tree-sitter* [17] to construct an Abstract Syntax Tree (AST); (2) We then identify the target AST nodes, including sink and jump nodes (serving as entry points for path analysis) as well as source nodes (serving as endpoints); (3) Starting from each entry point, we traverse the AST toward the root to collect all conditional nodes (i.e., *if* statements and *loop* statements detailed in Section 4.1.1) encountered along the path to the corresponding endpoint.

Agent Framework. We develop an agent framework based on the OpenAI API [14], which supports the definition of custom tools through its built-in tool-calling mechanism. This allows the LLM to autonomously trigger predefined *tool_calls* based on task requirements. We implement a custom tool named *search_context*, which retrieves relevant code blocks based on code element identifiers (e.g., function names, struct names, and macro names) returned by the LLMs. To support this functionality, we employ the *Lizard* library [13] to analyze source files and extract detailed function-level information, and use *ctags* [8] to identify and extract struct and macro definitions. The complete procedure for recursively exploring contextual functions to perform symbolic range reasoning is detailed in Algorithm 1.

Algorithm 1: Iterative Context-Aware Function Call Symbolic Range Reasoning

Input : Initial states P, Critical path conditional expressions $FPE(S_f, T_f)$, Maximum depth for nested function analysis max_depth

Output : Symbolic ranges of constraint-related function return values $\mathcal{R} = \{r_1, \ldots, r_k\}$

Procedure AnalyzeConstraintRelatedFunctions(P)
  $\mathcal{R} \leftarrow \emptyset$
  for *each function call expression* $fc \in FPE(S_f, T_f)$ **do**
    if `hasArguments(`fc, $P.variables$`)` **then**
      $f_{sig} \leftarrow$ `getFunctionSignature(`fc`)`
      $f_{body} \leftarrow$ `searchContext(`f_{sig}`)`
      $\mathcal{R} \leftarrow \mathcal{R} \cup$ `analyzeDeeperFunctionCall(`f_{body}, P, 1`)`
  return $\mathcal{R}$

Function f, P, $depth$
  if *not* `hasSufficientContext` *(*f*)* ▷ *decided by LLM agent*
  then
    $\mathcal{F} \leftarrow$ `identifyRequiredFunctions(`f`)` ▷ decided by LLM agent
    if $depth \leq max_depth$ **then**
      for *each* $f' \in \mathcal{F}$ **do**
        $f'_{body} \leftarrow$ `searchContext(`f'`)`
        $r_f \leftarrow$ `analyzeDeeperFunctionCall(`f'_{body}, P, $depth + 1$`)`
  else
    $r_f \leftarrow$ `symbolicRangeReasoning(`f, P`)`
  return r_f

Evaluation Setup. Except for RQ2 (Model Versatility), both PATHAGENT and the baselines use GPT-4o-mini [4] as the backbone LLM. To minimize randomness, we set the temperature to 0, ensuring the LLM performs greedy decoding without sampling. PATHAGENT and all studied baselines are tested on a Linux server (CPU: Intel Xeon 303 Platinum 8358P, GPU: NVIDIA A800 Tensor Core GPU).

C Prompt Design of PATHAGENT

C.1 Prompt in Context-Aware Symbolic Range Reasoning

Prompt for variable symbolic range reasoning:

> **System Role**: You are an expert C/C++ programmer.
>
> **Task Description:** Your goal is to determine the possible value ranges for each variable in [conditional statement], assuming the [P] when [conditional statement] is executed.
>
> Important:
>
> - Analyze each line sequentially through symbolic execution.
>
> - Consider function semantics based on their function names.
>
> - Track how variable ranges change through the execution path.
>
> ...
>
> **Code Snippet:** [Code Snippet]
>
> Please provide your analysis in clear steps, explaining how you arrived at the value ranges, and conclude with a summary of all variable ranges at the point of the [conditional statement].

Prompt for contextual analysis:

System Role: You are an expert C/C++ programmer.

Task Description: Given the function [function name] with body [function body], your goal is to determine the possible value ranges for the function return value when [P]. If the value range cannot be directly determined and further investigation in the deeper function is required, please use the search_context tool to retrieve the function body for further analysis.

Important:

...

Code Snippet: [Code Snippet]

Please provide your analysis in clear steps, explaining how you arrived at the value ranges, and conclude with a summary of the function return value range.

C.2 Prompt in SMT Query Script Generation

Prompt in SMT constraints generation:

System Role: You are an SMT constraint expert.

Task Description: Convert the following C/C++ conditional expressions/value ranges into Z3 SMT solver constraints: [Conditional Expressions]. When generating Z3 SMT solver scripts, follow these critical guidelines:

[SMT Query Script Generation Rules]

Return only the executable Z3 constraints in the following format:

```
‘‘‘python
var_name1 = Int(‘var 1’)
var_name2 = Bool(‘var 2’)
s.add(constraint 1)
s.add(constraint 2)
...’’’
```

Prompt in script merging:

System Role: You are an expert Z3 constraint solver.

Task Description: I’ll provide a Z3 constraint script and a set of additional constraints. Please combine them into a single script and keeping all existing constraints (both from the original script and the new constraint sets). If any new constraints are missing variable definitions, make sure to add or modify the necessary declarations. Return your solution as a function named *check_constraints()* that returns a boolean indicating whether the constraints are satisfiable. The function should be ready to execute and no imports or extra content needed.

Original Z3 script: [SMT Query Script]

Additional Constraint Set: [SMT Format Constraints]

Prompt in script fix:

Task Description: The Z3 script you generated contains errors. Please fix and regenerate the Z3 Python script. The script should be directly executable.

Error details: [Error Message]

Additional Requirements: [SMT Query Script Generation Rules]

D Experimental Results

Comparison among Bug Types. Table 7 compares PATHAGENT with four baselines on SAFP-BENCH-C across five bug categories. Notably, BOF, UAF and UBI contain no true positives. Consequently, precision and recall values cannot be calculated, and the precision for identifying false

positives (FPR_P) is consistently 1. Among the five bug types, the improvement for UFA and DBZ detection is limited due to the relatively simple nature of the cases, where false positives arise from significant validity checks preceding the critical operations on the target variable (i.e., use or reallocation after free, or division by zero), which do not involve complex context or path constraints.

Table 7: Comparison on Different Bugs

Type	Tech.	Acc.	FPR_P	FPR_R	Pre.	Recall
NPD	PATHAGENT	**0.75**	**0.98**	**0.72**	**0.36**	**0.93**
	LLM4SA	0.57	0.97	0.51	0.24	0.90
	LLMDFA	0.39	0.88	0.35	0.13	0.67
	RepoAudit	0.60	0.91	0.58	0.22	0.67
	IRIS	0.39	0.93	0.30	0.18	0.87
BOF	PATHAGENT	**0.86**	**1.0**	**0.86**	/	/
	LLM4SA	0.26	**1.0**	0.26	/	/
	LLMDFA	0.17	**1.0**	0.17	/	/
	RepoAudit	0.22	**1.0**	0.22	/	/
	IRIS	0.61	**1.0**	0.61	/	/
UAF	PATHAGENT	0.96	**1.0**	0.96	/	/
	LLM4SA	0.92	**1.0**	0.92	/	/
	LLMDFA	0.32	**1.0**	0.32	/	/
	RepoAudit	0.56	**1.0**	0.56	/	/
	IRIS	**1.0**	**1.0**	**1.0**	/	/
DBZ	PATHAGENT	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
	LLM4SA	0.94	**1.0**	0.93	0.67	**1.0**
	LLMDFA	0.65	**1.0**	0.65	0.0	0.0
	RepoAudit	0.69	0.85	0.79	0.0	0.0
	IRIS	0.63	0.90	0.64	0.17	0.50
UBI	PATHAGENT	**0.90**	**1.0**	**0.90**	/	/
	LLM4SA	0.76	**1.0**	0.76	/	/
	LLMDFA	0.41	**1.0**	0.41	/	/
	RepoAudit	0.46	**1.0**	0.46	/	/
	IRIS	0.02	**1.0**	0.02	/	/
Total	PATHAGENT	**0.80**	**0.99**	**0.79**	**0.34**	**0.94**
	LLM4SA	0.60	0.98	0.56	0.20	0.91
	LLMDFA	0.38	0.90	0.35	0.12	0.71
	RepoAudit	0.54	0.92	0.53	0.14	0.64
	IRIS	0.40	0.95	0.34	0.13	0.85

Comparison among Static Analyzers. Table 8 presents the effectiveness of PATHAGENT with three static analyzers in NPD detection. Among three analyzers, PATHAGENT demonstrates the greatest advantage in eliminating false positives from CodeQL (with a FPR_R improvement of 35% - 49%). This improvement is attributed to CodeQL's relatively comprehensive null pointer check rules, which lead to more complex false positives, such as the complex constraint cascade (e.g., Example-1 in Section 3), and require more extensive context analysis. Specifically, 44% of CodeQL cases involve context analysis, whereas only 0.29% and 0.36% of cases from cppcheck and infer involve context analysis, respectively.

Table 8: Comparison on Different Analyzers

Analyzer	Tech.	Acc.	FPR_P	FPR_R	Pre.	Recall
CodeQL	PATHAGENT	**0.75**	**0.97**	**0.71**	**0.42**	**0.91**
	LLM4SA	0.45	0.93	0.36	0.24	0.88
	LLMDFA	0.38	0.85	0.31	0.17	0.73
	RepoAudit	0.53	0.86	0.51	0.22	0.63
	IRIS	0.34	0.91	0.22	0.20	0.91
Infer	PATHAGENT	**0.68**	**1.0**	**0.63**	**0.30**	**1.0**
	LLM4SA	0.59	**1.0**	0.53	0.25	**1.0**
	LLMDFA	0.27	0.71	0.26	0.07	0.33
	RepoAudit	0.50	0.90	0.47	0.17	0.67
	IRIS	0.27	0.80	0.21	0.12	0.67
CppCheck	PATHAGENT	**0.79**	**1.0**	**0.77**	0.22	**1.0**
	LLM4SA	**0.79**	0.95	**0.77**	0.24	0.60
	LLMDFA	0.43	0.93	0.43	**0.54**	0.50
	RepoAudit	0.74	0.97	0.73	0.25	0.80
	IRIS	0.49	0.95	0.46	0.14	0.80

E Limitations

PATHAGENT incurs higher runtime than baselines, as it decomposes the reasoning process into multiple fine-grained sub-tasks. Future work may explore inference acceleration and parallelization to improve efficiency. The precision of PATHAGENT remains limited due to the extreme class imbalance in real-world software, where true bugs are rare. Nevertheless, PATHAGENT achieves significant improvements over static analyzers and prior LLM-based approaches. Future work could further improve precision through enhanced validation strategies. Our evaluation was conducted on C/C++ benchmarks, leaving the effectiveness of PATHAGENT on other programming languages unverified. However, since PATHAGENT is not specifically designed for any particular language, it can be readily adapted to other programming languages. In future work, we plan to broaden our benchmarks to cover a wider range of languages.

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes], [No], or [N/A].
- [N/A] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for [N/A]).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will also be asked to include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While [Yes] is generally preferable to [No], it is perfectly acceptable to answer [No] provided a proper justification is given (e.g., error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering [No] or [N/A] is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction accurately reflect the papers contributions and scope.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

Justification: The limitations are discussed in Appendix E

Guidelines:

- The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: There are no theoretical results in this paper.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We have provided all necessary information for reproduction in Appendix C and Appendix B

   Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Data and code of our work are provided in the Section 1 at https://github.com/LLM4PFA/LLM-Enhanced-Path-Feasibility-Analysis

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.

- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: This paper provides dataset construction details in the Appendix A. All implementation details are provided in the Section 5 and Appendix B.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: This paper provides statistical significance results on RQ1 in the Section 5.1

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The computer resources are provided in the Appendix B

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.

- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research conforms with the NeurIPS Code of Ethics in every respect.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: This paper prioritizes discussing technical methodologies and results rather than societal impacts.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: All models and datasets used in this study are publicly accessible. Thus, there is no risk for misuse.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: Data and code of our work are under MIT license.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: Our codebase contains a ReadMe providing thorough documentation.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [Yes]

Justification: These details are provided in the Appendix A

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [Yes]

Justification: These details are provided in the Appendix A

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: As a core component, this paper detailed the usage of LLMs in both the methodology and evaluation sections.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.