Towards More Accurate Static Analysis for Taint-Style Bug Detection in Linux Kernel

Haonan Li*, Hang Zhang†, Kexin Pei‡, Zhiyun Qian*
*University of California, Riverside, †Indiana University Bloomington, ‡The University of Chicago
*hli333@ucr.edu, zhiyunq@ucr.edu, †hz64@iu.edu, ‡kpei@cs.uchicago.edu

***Abstract*—Static analysis plays a crucial role in software vulnerability detection, yet faces a persistent precision-scalability trade-off. In large codebases like the Linux kernel, traditional static analysis tools often generate excessive false positives due to simplified vulnerability modeling and over-approximation of path and data constraints. While Large Language Models (LLMs) demonstrate promising code understanding capabilities, their direct application to program analysis remains unreliable due to inherent reasoning limitations.**

We introduce BUGLENS, a post-refinement framework that significantly enhances static analysis precision for bug detection. BUGLENS guides LLMs through structured reasoning steps to assess security impact and validate constraints from the source code. When evaluated on Linux kernel's taint-style bugs detected by static analysis tools, BUGLENS improves precision approximately 7-fold (from 0.10 to 0.72), substantially reducing false positives while uncovering four previously unreported vulnerabilities. Our results demonstrate that a well-structured, fully-automated LLM-based workflow can effectively complement and enhance traditional static analysis techniques.

I. INTRODUCTION

Static analysis has long served as a cornerstone technique for identifying software vulnerabilities. These techniques aim to detect various security weaknesses, such as buffer overflows and information leaks. However, static analysis tools often struggle to balance the trade-off between precision and scalability [1], [2].

More precise analysis, *e.g.,* symbolic execution [3], can be computationally expensive and often infeasible for large codebases such as the Linux kernel [4]. Conversely, more scalable techniques sacrifice the precision for scalability, leading to a high number of false positives. For example, Suture [5], an advanced taint bug detection in the Linux kernel, shows a 90% raw false positive rate, incurring substantial manual effort to inspect its results.

Specifically, the imprecision of existing static analysis approaches stems from the following two key issues:

- **Simplified Vulnerability Modeling.** Static analyzers often rely on simplified *heuristics* for vulnerability detection. For example, a static analyzer may flag every *arithmetic operation* as potentially overflowing. While this simplification might ensure no genuine vulnerabilities are missed, it inflates the number of false positives.
- **Over-Approximation of Path and Data Constraints.** To avoid exponential path exploration, static analyzers often make coarse assumptions about whether a path is feasible or how data flows through the program. While such an over-approximation ensures the analysis is completed in a reasonable time, it also flags numerous *infeasible* paths as potentially vulnerable, resulting in excessive false positives.

Recent advances in *Large Language Models* (LLMs) offer a promising avenue for overcoming these issues. Trained on vast amounts of code and natural language, LLMs exhibit remarkable capabilities in understanding code semantics, API usage patterns, and common vulnerability types [6]–[8]. With these broader insights, the LLM-based approach can potentially: (1) *enhance vulnerability modeling* by providing a more nuanced understanding of code semantics, and (2) *refine path and data constraints* by selective analysis of semantically plausible paths and data flows.

However, LLMs are *not a silver bullet* for program analysis. Despite their approximate semantic understanding capabilities, LLMs are not inherently equipped for the rigorous demands of program analysis [9], [10]. Their reasoning proves brittle, particularly when confronted with the complex program dependencies crucial for security analysis [11]–[13]. Our initial experiments also confirm that naively applying LLMs to program analysis, for instance, by simply asking *"Does this static analyzer report an actual bug?"*, yields highly unreliable results, frequently misclassifying vulnerabilities and failing to identify critical flaws. This is often because LLMs tend to fixate on surface-level code features, missing the critical dependencies that dictate program behavior and security properties, especially within intricate control and data flows.

In this work, we introduce BUGLENS, an innovative framework that *post-refines* the results of static analysis using LLMs. Rather than blindly applying LLMs to analyze programs, BUGLENS is carefully orchestrated to teach LLMs key concepts of program analysis and guide them toward reasonable analytical procedures. By analyzing the output of static analysis, BUGLENS complements the limitations of existing tools, especially in terms of precision, and yields more accurate and actionable vulnerability detection for practical codebases.

Our key approach is to introduce a structured framework to guide LLM's reasoning on the code. It decomposes the reasoning processes into a series of guidelines following the traditional program analysis workflow. By constraining the LLM reasoning space within this established methodology, the model is grounded with a more rigorous reasoning scaffold than it would follow by default, thus mitigating the inherent limitations of LLMs in code reasoning.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00039

We demonstrate that this combined approach significantly improves the precision of taint-style bug detection in the Linux kernel, reducing the need for manual inspection of false positives and even uncovering previously ignored vulnerabilities. We summarize our contributions as follows:

- **Post-Refinement Framework.** We introduce BUGLENS, an LLM-based framework that complements static analysis to boost its precision, overcoming various practical weaknesses identified from real-world complex Linux Kernel codebases.
- **Structured Analysis Guidance (SAG).** We design a structured reasoning workflow that decomposes the LLMs' reasoning into a series of rigorous steps for reliable predictions.
- **Empirical Results.** BUGLENS improves the precision of state-of-the-art static analysis tool on the Linux kernel from 0.1 to 0.72. Significantly, BUGLENS also corrects four false negatives that prior manual analysis incorrectly filtered.
- **Open Source.** We open-source our implementation to facilitate future research on LLM-augmented program analysis. The code and data are available at https://github.com/seclab-ucr/BugLens.

II. BACKGROUND

A. Taint-Style Bugs in the Linux Kernel

Taint-style bugs involve insecure data propagation, where unsanitized data (taint *source*) reaches a sensitive program location (taint *sink*), causing flaws like out-of-bound access. For instance, an unchecked input integer (source) might become an array index (sink). Proper *sanitization* (*e.g.,* range checks) is typically missing or insufficient. Such bugs can lead to severe vulnerabilities like buffer overflows or denial-of-service.

In the kernel, these bugs often occur when untrusted user input (*e.g.,* syscall arguments) propagates to sensitive kernel operations (*e.g.,* arithmetic calculations) without adequate sanitization [5], [14]. Detecting these bugs is challenging due to:

- **Diverse and General Sinks.** Unlike user-space applications with often well-defined sink APIs (*e.g.,* `exec()`), the kernel's high privilege means almost *any* operation (even a single arithmetic calculation) could be a sink if affected by unsanitized user data. This broad scope increases false positives. Additionally, low-level coding optimizations common in the kernel (*e.g.,* intentional integer overflows) can complicate accurate detection.
- **Scattered and Intricate Sanitization.** Sanitization checks can be distant from sinks, often crossing function boundaries within the large and complex kernel codebase. These checks are frequently intertwined with domain-specific kernel logic (*e.g.,* privilege systems, configurations), posing significant hurdles for taint analysis tools.

B. Static Analysis Tools for Taint-Style Bugs in Kernel

Suture [5] is the state-of-the-art work from academia aiming at kernel taint-style bug detection, which extends the previous work, Dr. Checker [14], by adding cross-entry taint tracking capability and improving analysis precision. CodeQL [15] is one of the most powerful and widely used industry code analysis engines, capable of taint analysis [16] and query-based bug detection based on customizable rules. CodeQL has a large community that develops and maintains many different bug detection rules, including those for kernel taint-style bugs (*e.g.,* [17]). Despite their effectiveness, all these tools exhibit high false alarm rate (*e.g.,* ~90%), underscoring the inherent difficulties in accurately identifying kernel taint-style vulnerabilities. We elaborate on these difficulties and challenges in §III.

III. CHALLENGES AND DESIGN RATIONALE

A. Challenge 1 (C1): Simplified Vulnerability Modeling

The first source of imprecision is the reliance on *simplified vulnerability detection* modeling. To maintain scalability and avoid missing potential bugs, static analyzers often employ overly simplified detection rules, especially in taint-style bug detectors.

For instance, a static analyzer typically flags specific code patterns as potential bugs. A *Tainted Arithmetic Detector (TAD)* would identify any arithmetic operation involving tainted variables (*e.g.,* `var += size`, where `var` is tainted) as a potential integer overflow vulnerability. However, in Linux kernel practice, these patterns frequently do not translate to exploitable vulnerabilities for two key reasons:

- **The Behavior Itself is *Benign***. The kernel's reliance on *low-level C idioms, pointer manipulation, and intentional 'unsafe' design patterns* often confounds traditional analyzers, leading to inaccuracies, e.g., our evaluation shows that 61.2% of the flagged 'bugs' were actually benign (§VI-C2). For example, in Linux drivers, there are cases where integer overflow is expected and benign, such as with `jiffies`:

```
unsigned long start = jiffies;
// do something time-sensitive...
if (time_after(jiffies, start + timeout)) {
  ...
}
```

  For these cases, even when integer overflow is possible due to the tainted variable `timeout`, it is not a security vulnerability, as `time_after` is defined as `#define time_after(a, b) (b - a) > 0`). Such false positives commonly arise from intentional design patterns in the Linux kernel, including struct hacks (deliberate out-of-bounds access), type casting, unions, and data structure operations (*e.g.,* `ptr->next`, `container_of`).
- **The Tainted Data is Properly Validated** before reaching the sink. For example, the *Taint in the Loop Bound* (TLB) pattern is common in the Linux kernel, but many include proper validation. Consider this simple array traversal:

```
assert(nums < sizeof(arr) / sizeof(int));
for (i = 0; i < nums; i++) { run(arr[i]); }
```

  Without the `assert` check, this loop could lead to an out-of-bounds access. However, the `assert` ensures that the loop index never exceeds the array bounds, making it safe. Static analyzers that does not account for such checks (*e.g.,* path-insensitive ones) will misclassify this as a vulnerability, leading to false positives.

B. Challenge 2 (C2): Over-Approximation of Path & Data Constraints

As mentioned above, the tainted data is often checked before the sink, and with a safe range. However, static analysis tools often *over-approximate* the path and data constraints to ensure scalability, as the kernel's **immense scale and continuous evolution** make precise static analysis impractical, and the logic of the sanitization could often be distant or complex. Alternatively, a common heuristic in practice [18] is to directly prune paths with checks, regardless of their actual effects. However, the existence of checks is not always effective in preventing the bug from happening, *e.g.,* we discovered some unreported bugs due to a plausible but ineffective check (see §VI-B). A simplified example is shown in the following, which is designed to sanitize the tainted variable `num`:

```
int clamp_num(int config, int num){
  if (config && num > ARR_SIZE)
      num = ARR_SIZE;
  return num;
}
```

The function `clamp_num` *only* sanitizes the tainted variable `num` effectively when `config` is set.

Even though this example is relatively simple, many others involve complicated data relationships and path constraints in the Linux kernel. They often appear together with more complex constructs, including loops and advanced data structures (see §VI-E). This combination makes precise analysis (both precise value range (`num > ARR_SIZE`) and path-sensitive (only effectively when `config`)) difficult.

C. Challenge 3 (C3): Reasoning Hurdles for LLMs

LLMs have shown great promise in solving C1 and C2 [19]–[22]. Trained extensively on vast amounts of Linux kernel code, LLMs can recognize C idioms and intentional kernel design and can thus identify benign patterns that static analyzers often misclassify as bugs (for C1). Besides, LLMs are often aware or able to identify the *intent* of the Kernel function based on the hints from other modalities, *e.g.,* a sanitization check often named with `check_` or `validate_` prefix is a typical natural language hint, in addition to the comments inserted by the Kernel developers. As a result, the LLM-based analysis can often quickly narrow down to a small amount of code to facilitate a more nuanced analysis with semantic understanding (for C2).

Recent works [19], [23] also show that LLMs can be used to refine static analysis results (C1 & C2) in a straightforward manner, *i.e.,* to directly ask LLM with *"Is this case from static analysis report vulnerable?"* followed by the code snippet. However, our experiments (detailed in §VI-C) demonstrate that such a simple prompting strategy leads to high false negatives — cases where the LLM incorrectly classifies actual vulnerabilities as safe code, especially against the presence of complex control flow and data constraints.

We observe that the false negatives stem from the fundamental limitation of existing LLMs when tasked with sophisticated, structured, and symbolic reasoning. Specifically, with simple prompting, LLMs often rely on surface-level spurious features [24] to reason about the vulnerability. For example, the presence of a *sanity check* often correlates statistically with *safe code* (many practical static analyzers use such heuristics to prune paths). We observe the model classifies code containing such a check as safe without performing the deeper reasoning required to determine *if the check is actually effective under the current execution* paths. Real-world vulnerabilities often exploit exactly these scenarios: checks that are bypassable, incomplete, or rendered ineffective by intricate control and data flows. The model's tendency to rely on spurious patterns without performing the actual reasoning has been shown ineffective in tracking critical dependencies and results in unreliable predictions in many different domains [25].

D. Design Rationale

To address challenges C1 through C3, we introduce BUGLENS, a fully automated, multi-stage LLM-based framework designed specifically as a post-refinement layer for static analyzers. The key design philosophy of BUGLENS is to scale the test-time compute of LLMs by allocating more tokens for precise reasoning of the intended program behaviors based on the model's understanding of program semantics [26]. Specifically, BUGLENS consists of the following specific key components, each targeting a specific challenge:

- **Security Impact Assessor (SecIA)**: *Addressing Simplified Vulnerability Models (C1).* Instead of relying solely on predefined patterns, BUGLENS leverages LLMs to analyze the *potential security impact* if tainted data identified by static analysis were completely controlled by an attacker. It then evaluates whether the tainted value could potentially lead to security vulnerabilities (*e.g.,* memory corruption, denial of service (DoS)), as detailed in §IV-B. By focusing on semantic consequences rather than simplified vulnerability modeling, SecIA provides a more precise assessment of security impact.
- **Constraint Assessor (ConA)**: *Addressing Over-Approximation of Path & Data Constraints (C2).* ConA uses LLMs to analyze whether data constraints in the code are sufficient to prevent potential vulnerabilities from being triggered. By tracing how tainted data is processed and constrained, ConA performs heuristic reasoning (*e.g.,* to find sanitizations with names `check_x`) to evaluate the effectiveness of these safeguards. This approach is designed to handle the complexity and scalability challenges of systems like the Linux kernel, where traditional formal methods may lack precision or scalability.
- **Structured Analysis Guidance (SAG)**: *Addressing Reasoning Hurdles for LLMs (C3).* As described in §III-C, simply prompting LLMs for analysis often leads to spurious reasoning. To this end, SAG grounds LLMs' reasoning in ConA with the scaffold that describes the typical program analysis workflow. At a high level, SAG employs specific prompts to describe key analysis principles and include in-context examples to demonstrate how to systematically dissect code, trace dependencies, and evaluate conditions,

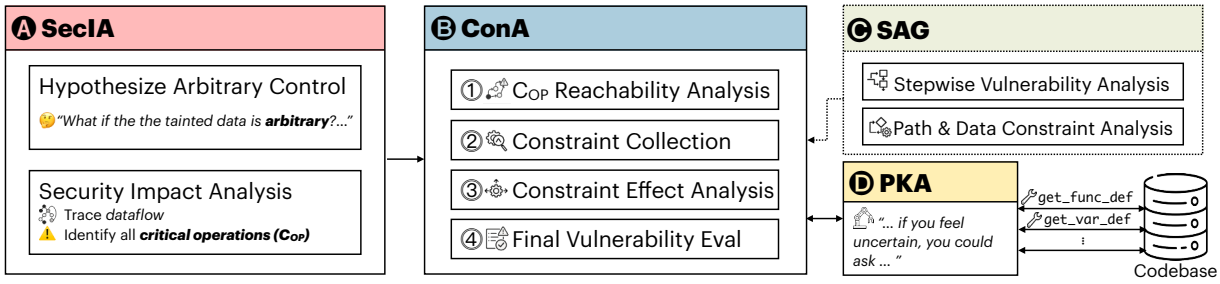

Figure 1: Overview of BUGLENS, showing (A) Security Impact Assessor (SecIA) first assesses the security impact of the potential bugs identified by static analysis, and (B) Constraint Assessor (ConA) assesses data constraints and evaluates if the bug is feasible. ConA is guided by (C) Structured Analysis Guidance (SAG) to reason about code more effectively, and can interact with (D) Project Knowledge Agent (PKA) to get information about the codebase on demand.

especially in complex scenarios. This guidance constrains the LLM towards a more rigorous and reliable analysis process. §IV-D elaborates on the design of SAG.

IV. DESIGN

BUGLENS first takes the static analysis report as input, which identifies the context of the potential bugs and the tainted data flow, and then performs the following steps to evaluate the potential bugs, as shown in Figure 1:

- **Security Impact Assessor (SecIA)**: This component evaluates the security impact of the potential bugs identified by static analysis. It identifies the Critical Operations (C_{OP}) that are influenced by the tainted data and classifies them as either *Normal Code* or *Requires Constraint Analysis*.
- **Constraint Assessor (ConA)**: This component performs a multi-step analysis to evaluate the feasibility of the potential bugs. It collects the path conditions and data constraints, summarizes them, and evaluates whether they are effective in preventing the vulnerability.
- **Structured Analysis Guidance (SAG)**: This component provides the scaffold to constrain the LLM through predefined code reasoning procedures to help it understand the code and analyze the constraints effectively.
- **Project Knowledge Agent (PKA)**: This component allows the LLM to access the codebase on-demand, enabling it to retrieve global codebase information.

Additionally, BUGLENS also adopts the common prompting techniques to scale the test-time compute, such as (1) Majority-vote querying, where we query the model multiple times and take the most common answer; (2) Chain-of-Thought (CoT) prompting [27]; and (3) Schema-constrained summarization, where a follow-up prompt requests the model's own output in a predefined XML format, making LLM's response easy to parse.

A. Premises

In BUGLENS, we design a framework that leverages LLMs to refine the results of static analysis for bug detection. We focus on the case of taint-style bugs, where a vulnerability arises from the flow of untrusted or tainted data.

As input, we assume that a static analysis tool has already performed taint propagation and produced a set of candidate warnings. In particular, the tool flags certain program locations as *sinks*, which are critical operations that may cause vulnerabilities if they involve tainted values (*e.g.,* pointer dereference). These candidate sinks (referred as C_{OP} in the following sections), together with the context provided by static analysis (*e.g.,* taint traces), are the primary input to our system.

The expected output of our framework is a refined classification of these candidate bug reports. Specifically, the LLM is used to filter out false positives and to provide a more semantically grounded explanation of whether a flagged sink represents a real bug. We emphasize that BUGLENS does not aim to replace static analysis. Instead, it operates on static analysis results, refining them to improve precision.

B. Security Impact Assessor (SecIA)

The core insight behind *SecIA* is based on the fundamental evaluative question: *What are the consequences if tainted data assumes arbitrary values?*

1) Core Assumption and Rationale: SecIA operates on a fundamental assumption regarding attacker capability at the point of initial impact assessment: ***Aribitary Control Hypothesis (AC-Hypo)***. For a given program location K where static analysis reported an operation $C_{OP}(v)$ involving tainted data v (the *sink*), (1) we hypothesize that an attacker can control v to take any value v_{atk} (within the constraints of its data type), and (2) we *provisionally ignore* any effects of checks or path conditions (even explicit checks) encountered on analysis. In other words, we assume that the attacker can take *any* value to anywhere (successors of the sink node in the control flow graph).

This hypothesis enables SecIA to streamline analysis. SecIA focuses solely on *potential security impact*. This permits *early filtering* of findings based purely on the consequence, (safely) reducing subsequent analysis load. Critically, this approach *defers* the complex analysis of actual program constraints—including path feasibility, value ranges, and importantly, *whether those protective checks or sanitizers* really work. This deferral *mitigates false negatives* (FNs) by preventing premature dismissal of vulnerabilities due to reliance on potentially bypassable checks or inaccurate LLM constraint reasoning about their effectiveness. The effectiveness is shown in §VI-C2.

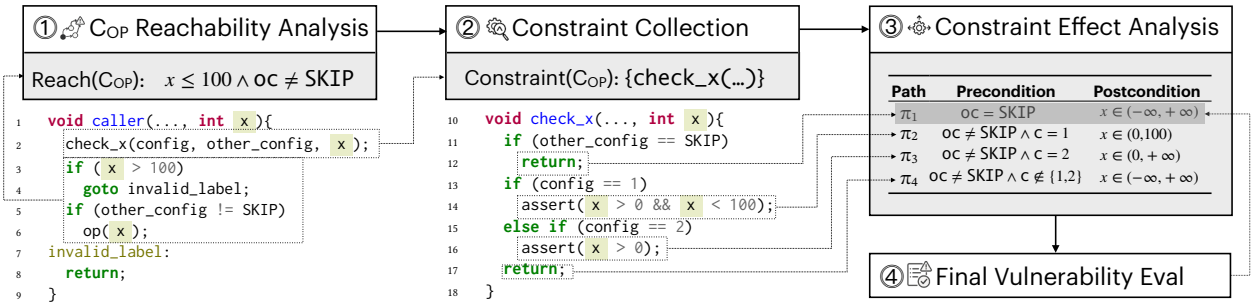

Figure 2: The workflow of Constraint Assessor (ConA) with an example, where the `x` is tainted and `op(x)` is the critical operation (C_{OP}). The sanitization function `check_x(..., x)` is affected by the `config` (noted as `c`) and `other_config` (noted as `oc`).

2) Security Impact Analysis: SecIA performs a *forward influence analysis* to identify the influenced critical operations (C_{OP}) that are potentially affected by the tainted data v from the sink location K. It then filters out the benign operations that are not security-sensitive based on whether this operation could (potentially) result in memory bugs (*e.g.,* out-of-bound access, arbitrary memory access) or DoS. For instance, the `jiffies` case in §III-A will be attributed as "not a bug".

C. Constraint Assessor (ConA)

The *Constraint Assessor (ConA)* aims to identify whether the bug of C_{OP} can be triggered by analyzing the data constraints. As shown in Figure 2, the Constraint Assessor involves a four-step workflow:

- **Step 1: Critical Operation Reachability Analysis.** The process begins by determining the conditions under which program execution can reach the specific *Critical Operation (C_{OP})* location. This establishes the base requirements for the vulnerability to be possible.
- **Step 2: Constraint Collection.** Next, the analysis traces the tainted data flow path(s) backward from the C_{OP} towards the data's source. Along this path, it identifies code segments—such as conditional statements, assertions, or calls to validation functions—that appear intended to act as *constraints* on the tainted data's value or range before it is used at the C_{OP}.
- **Step 3: Constraint Effect Analysis.** Each potential constraint identified in Step 2 is then analyzed in detail. This step aims to understand the constraint's specific *effect*: Under what conditions (*precondition*) does the constraint need to satisfy, and what impact does it have on the tainted variable's possible numerical range (*postcondition*)?
- **Step 4: Final Vulnerability Evaluation.** Finally, it performs an evaluation to determine if the potential vulnerability can still be triggered. If yes, the case is classified as a *"Potential Vulnerability."* Otherwise, if the LLM determines that the constraints effectively prevent the vulnerability from being triggered, the case is classified as *"Eliminated."*

This workflow leverages LLMs to interpret code (mimicking a formal reasoning process), identify relevant patterns and constraints, and perform the reasoning required for each step.

We employ this LLM-based approach acknowledging the trade-offs of soundness and precision. ConA aims to overcome the potential precision limitations of conservative abstractions and achieve broader applicability in the complex Linux kernel. In §VI-B and §VI-D, we thoroughly examine how our design in relaxing the strict over-approximation introduces only a reasonably small number of false negatives. In the following sections (§IV-C1 through §IV-C4), we elaborate on the specific design, rationale, heuristic considerations, and soundness implications inherent in each stage of this analysis process.

1) Step 1: Critical Operation Reachability Analysis: The analysis begins by determining the conditions required for program execution to reach the specific *Critical Operation* (C_{OP}) location previously identified by SecIA as potentially vulnerability. These reachability conditions form the base constraints that must be satisfied for the vulnerability to be triggerable via this C_{OP}.

Condition Representation: When analyzing path conditions, we allow the LLM to generate natural language summaries that capture the semantic meaning of complex reachability conditions. For instance, where traditional analysis might struggle, an LLM might identify a condition like: *"The operation is only executed if `init_subsystem()` returned zero (success), AND the device state in `dev->status` equals `STATUS_READY`."*

The LLM is later asked (in Step 4) to evaluate these semantic constraints to assess the potential impact. The natural language representation allows the LLM to leverage its understanding of the code's intent and context but introduces a higher degree of uncertainty in the final evaluation, which we must acknowledge.

2) Step 2: Backward Constraint Collection: Once the Critical Operation (C_{OP}) and its local reachability conditions are identified, the next step is to gather potential constraints imposed on the tainted data before it reaches the C_{OP}. This involves tracing the data flow path(s) for the tainted variable backward from the C_{OP} toward its source(s).

Example: In Figure 2, tracing back from `critical_op(x)` in `caller()`, the LLM identifies the call `check_x(...)` to be a potential constraint on `x`. It would query the PKA for the function definition and analyze its effect, as detailed in Step 3.

3) Step 3: Constraint Effect Analysis: After collecting potential constraining code segments (like the function `check_x()`) in Step 2, this step analyzes the effect of these segments on the tainted variable. The goal is to understand how different execution paths within these segments modify the possible range of the tainted variable and under which conditions (*preconditions*) those paths are taken.

This step prompts the LLM to first identify all major execution paths through the provided code segment (*e.g.,* `check_x()`), and then considers the preconditions and post-conditions (data constraints of the tainted data) of each path.

Example: In Figure 2, we ask the LLM to analyze the function `check_x()` to summarize its effects on the tainted variable `x`. Particularly, the path π_1 and π_4 are ***bypass*** paths, and `check_x()` will not effectively limit the range of `x` with these preconditions.

4) Step 4: Final Vulnerability Evaluation: This final step synthesizes the precondition and postcondition from the previous analyses to determine if the identified constraints effectively neutralize the potential vulnerability associated with the Critical Operation (C_{OP}). And then it classifies the reported vulnerability as either "*Eliminated*" (constraints effectively prevent the vulnerability happening) or "*Potential Vulnerability*" (no effective constraints).

Example: In Figure 2, Step 1 of the reachability analysis shows that `oc` $\neq$ `SKIP`, and therefore the path π_1 can be eliminated, as its precondition is `oc` $=$ `SKIP`. Since `c` and `oc` are unknown, the function `check_x()` does not impose additional range constraints (we can only assume path π_4 to be valid with conservative analysis). The range analysis for `x` yields $(100, \texttt{INT\_MAX})$, and the final evaluation requires an understanding of `op()` itself. If the range is sufficient to prevent the bug, the issue is considered *eliminated*.

D. Structured Analysis Guidance (SAG)

To support the fine-grained code reasoning steps in ConA, we employ Structured Analysis Guidance (SAG) to scale the test-time compute of LLMs with structured reasoning templates and few-shot examples. At a high level, SAG grounds the reasoning procedures of LLMs with typical program analysis steps by eliciting more reasoning tokens during inference. Specifically, SAG assists Constraint Assessor (ConA) with the following two types of analysis: (i) *Guided Stepwise Vulnerability Analysis* with step-by-step instructions that decompose the analysis of precondition and postcondition (as described in §IV-C), and (ii) *Guided Path Condition and Data Constraint Analysis* to demonstrate how to analyze challenging data constraints and path conditions from code.

1) Guided Path Condition Analysis: In the analysis of the path condition in Step 1 and Step 3 of ConA (§IV-C1 and §IV-C3), we guide the LLM using prompts designed to extract path conditions from the source code surrounding the *operation* of interest. For example, consider the path leading to the C_{OP}(`op(x)`) within the `caller` function:

```
void caller(int config, int other_config, int x){
  check_x(config, other_config, x);
  if (x > 100)
    goto invalid_label; // skip the C_OP if true
  if (other_config != SKIP)
    op(x);   // reach the C_OP
invalid_label:
    return;
}
```

SAG asks the LLM to identify and categorize:

- **Bypass Conditions:** Identify conditional statements where taking a specific branch avoids the operation. The LLM is instructed to extract the condition and negate it to find the requirement for not bypassing the operation.
  Example: The condition `x > 100` leads to `invalid`, bypassing `critical_op(x)`. The negated condition required to proceed towards the OP is `x` $\leq$ `100`.
- **Direct Conditions:** Identify conditional statements where taking a specific branch is necessary to reach the operation along the current path. The LLM extracts the condition directly.
  Example: Reaching `sink(x)` requires entering the `if` block, so the condition is `other_config` $\neq$ `SKIP`.

The LLM then combines these conditions using logical `AND` to form the path-specific reachability constraint set for the operation. For this path in the example, the derived reachability condition is: $x \leq 100 \wedge$ `other_config != SKIP`.

2) Guided Data Constraint Analysis: In the analysis of data constraints in Step 2 and Step 3 of ConA (§IV-C2 and §IV-C3), particularly, we prompt LLMs to focus on the following data constraints:

- **Type constraints.** The variable's static type already restricts its range (*e.g.,* `uint8` is always in the range of $[0, 255]$).
- **Validation (*transferable* to source).** The program *tests* the value and aborts or reports an error if the test fails, *without modifying the value*. Because the check refers to the *current value*, the knowledge gained from this check (*e.g.,* "the value must be ≥ 0 on the success branch") also applies to *all source variables that influenced this value in the data flow*.
- **Sanitization (*not transferable* to source).** The program *writes a new, corrected value* back to the variable (*e.g.,* clamping it to a range). This operation serves the connection to the original value, so any property we learn afterward applies only to the sanitized copy, not to the original source variables in the data flow.

The key difference is that *validation can travel backward along the data-flow graph*, while sanitization overwrites the data and stops the transfer. Considering the folllowing example, suppose `v` is tainted:

```
int u  = v + 1;  // u is also tainted by v
if (u < 0)     // (1) validation
    return -EINVAL; // succeeds only if u >= 0,
                    // therefore v >= -1
u = clamp(u, 0, 100); // (2) sanitization
                // now u is guaranteed 0..100,
                // but not affect v
return use(u, v);
```

Step (1) is a *validation*: it *reads* `u` and branches, so the fact '`u` ≥ 0' (hence `v` ≥ -1) becomes part of the path

condition and is *transferable* to other variables in earlier nodes (`u = v +1`). Step (2) is a *sanitization*: it *writes* a new value into `u`; the constraint "$0 \leq$ `u` ≤ 100" holds only *after* this assignment. It is worth noting that the sanitization to `u` does not pose any constraints to `v`. However, if we replace the `clamp()` with an `assert(u< 100)`, we would get a constraint of `v` as well, `v` < 99.

Formal program analysis naturally distinguishes between validation and sanitization; however, LLMs often confuse these concepts (*e.g.,* missing transferred validation, treating sanitization as validation). SAG emphasizes these differences with few-shot examples, and therefore improves the LLMs' reasoning on data constraints.

E. Project Knowledge Agent (PKA)

BUGLENS leverages Project Knowledge Agent (PKA), a simple LLM code agent equipped with custom tools to automatically navigate the codebase and retrieve the related code context during the analysis. It starts the analysis with the context of the sink and explicitly requests more context (*e.g.,* function definitions `get_func_def`, global variable definition `get_var_def`, struct layouts) if needed. PKA then retrieves the relevant code snippets iteratively until the model self-decides that it has sufficient context to complete the analysis.

We provide a set of request types, enabling the LLM to gather broader codebase information at any point. This design is both flexible and extensible: new request types can be supported by adding corresponding backend callbacks. PKA searches the relevant code snippets based on CodeQuery [28]. We implement PKA in 500 lines of Python code, and it is agnostic to the underlying LLM model.

V. IMPLEMENTATION

BUGLENS is implemented with ≈7k tokens in prompts. The prompts are multi-turn dialogues, where it starts with a system message and the output of static analysis, followed by several interactions and tools (*e.g.,* PKA) to guide the LLM to perform the tasks step by step, as we describe in §IV. The detailed prompts can be found in our artifact.[1]

Below we describe several key implementation principles for each component of BUGLENS.

A. SecIA

The key idea is to let the LLM act as a semantic filter over the raw reports produced by the static analysis tool. We design a detailed prompt with examples that asks the LLM to classify each candidate report as either a *Potential Bug* or *Normal Code*, based on its understanding of the code snippet and the nature of the tainted variable.

- **Input:** Each prompt takes the flagged *code snippet*, the suspected *tainted variable*, and the *bug category* reported by the static analysis tool.
- **Output:** The model classifies each case as either *Potential Bug* or *Normal Code*, with an explanation that can be inspected by further analysis.

[1] https://github.com/seclab-ucr/BugLens-Code/blob/main/prompts/request.yaml

B. ConA

While the initial prompt design identifies potential bugs, it does not consider whether sanitization or other constraints may eliminate the risk. To address this, we introduce ConA, which refines the analysis by reasoning about the conditions under which a flagged sink can actually lead to an exploitable bug.

- **Input:** ConA takes as input the candidate bug reports from static analysis (and our previous stage) together with contextual information such as the *tainted variable*, its *propagation chain*, and *surrounding code*.
- **Output:** ConA provides a more accurate range analysis and classification: not only flagging dangerous uses of tainted data, but also distinguishing cases where existing checks already ensure safety.

C. Schema-constrained Summarization

At the end of each stage, we use a schema-constrained summarization prompt to convert the LLM's free-form text output into a structured XML format. This makes it easy to parse and use in subsequent stages of the pipeline. For example, at the end of SecIA, we prompt the LLM to summarize its findings in the following XML schema:

```
<tainted_var>tainted_var</tainted_var>
<vuln>
  <type>out_of_bound_access</type>
  <desc> an out-of-bound access at code
  ↪  `arr[tainted_var]`</desc>
</vuln>
```

VI. EVALUATION

Our evaluation aims to address the following research questions.

- **RQ1: (Effectiveness)** How effective is BUGLENS in identifying vulnerabilities?
- **RQ2: (Component Contribution)** How does the the prompt design affect the performance of BUGLENS? especially for the SecIA and SAG component?
- **RQ3: (Model Versatility)** How does the performance of BUGLENS vary across different LLMs?

A. Experimental Setup

We primarily evaluate BUGLENS using OpenAI's o3-mini model (`o3-mini-2025-01-31`). This model was chosen as our primary focus because, as demonstrated in our evaluation for RQ3 (Section VI-D), it achieved the best overall performance on our bug analysis task compared to several other recent leading models.

To address RQ3 regarding the generalizability of BUGLENS, we also tested its performance with a range of prominent alternative models, encompassing both closed-source and open-source options. These include: OpenAI's o1 (`o1-2024-12-17`), GPT-4.1 (`gpt-4.1-2025-04-14`), Google's Gemini 2.5 Pro (`gemini-2.5-pro-preview-03-25`), Anthropic's

Claude 3.7 Sonnet (`claude-3-7-sonnet-20250219`), and the open-source DeepSeek R1 (671B) model. This selection allows us to assess how BUGLENS performs across different model architectures, sizes, and providers. (*Note:* All experiments were conducted in Apr 2025, using the latest versions of these models available at that time.)

1) Cost and Performance: In our current prototype using the OpenAI o3-mini LLM backend, the analysis of 120 cases took approximately **4 hours** and cost **$8.62 in total** (or about **$0.07 per case** on average). This corresponds to:

- ≈**40.9K input tokens** and ≈**9.6K output tokens** per case (≈**50.5K total tokens**),
- ≈**2 minutes** of processing time per case (without parallelization).

This model (o3-mini) was chosen as our primary focus because, as demonstrated in our evaluation for RQ3 (Section VI-D), it achieved the best overall performance on our bug analysis task compared to several other recent leading models.

2) Evaluation Dataset: Linux Kernel Driver Analysis with static analyzers: Our study utilizes the Android kernel (Linux version 4.14.150, Google Pixel 4XL), which served as the testbed in the original Suture study [5]. For our initial analysis, potentially informing RQ1 regarding baseline performance and the challenges of automated bug detection in this complex environment, we applied prior static analysis tools, Suture and CodeQL-OOB, to the Linux kernel device drivers. The key findings from this analysis provide important context:

- **Suture:** When applied to the Linux device drivers, Suture initially generated 251 potential bug reports. This raw output translates to a high False Positive (FP) rate of approximately 90%. Suture employed a subsequent semi-automated refinement process, reporting a *reviewer-perceived* FP rate of 51.23%. However, this figure relies on Suture's broader definition of a bug, which classified all integer overflows as true positives. Furthermore, during our investigation, we identified 4 additional instances, initially dismissed as FPs by Suture authors' verification, that were indeed real bugs (by either their standard or ours). This finding highlights potential inconsistencies in large-scale manual verification efforts.
- **CodeQL-OOB:** We leverage CodeQL [29] as a complementary static analysis tool beyond Suture. Based on Backhouse et al.'s approach [17], we implemented a simple inter-procedural taint tracking analysis (CodeQL-OOB) that traces data flows from `ioctl` entry points to pointer dereference and `copy_from_user` (two typical cases of OOB bugs) to detect potential stack overflows. Running CodeQL-OOB on the Linux device drivers, it yields 24 potential bug reports. Our manual analysis confirmed 1 of these as a true positive (consistent with the known stack overflow bug reported in [17]). This corresponds to an FP rate of 95.8%.

3) Cost: On average, the cost of running BUGLENS is about $0.1 per case, under the latest version of OpenAI o3-mini. Each case takes about a few minutes to complete.

B. RQ1: Effectiveness

Table I. Performance of BUGLENS on top of Suture and CodeQL-OOB.

Method	TP	TN	FP	FN	Prec	Rec	F_1
Suture	24	0	227	0	0.10	-	-
$\text{Suture}_{\text{RP}}$	20	202	25	4	0.44	0.83	0.58
$\text{Suture}_{\text{BUGLENS}}$	24	218	9	0	0.72	1.00	0.84
CodeQL-OOB	1	0	23	0	0.04	-	-
$\text{CodeQL-OOB}_{\text{BUGLENS}}$	1	22	2	0	0.33	1.00	0.5

1) Precision and Recall: Table I shows the performance for the evaluated two static analyzers and our post-refinement method, BUGLENS. The results show BUGLENS significantly enhances precision of both Suture (0.10) and CodeQL-OOB (0.04). For CodeQL-OOB, BUGLENS increases precision substantially to 0.33 while not missing any real bugs detected before. For Suture, the precision is increased to 0.72 due to a drastic reduction in false positives (from 227 to 9). This refinement does not miss any exisiting bugs.

Moreover, noting that the semi-automated method in Suture, noted as $\text{Suture}_{\text{RP}}$, actually shows a lower recall (0.83) than the BUGLENS refinement (1.0). This is because after examining the positive results of $\text{Suture}_{\text{BUGLENS}}$, we found 4 cases of real bugs that were incorrectly classified as false positives during the manual inspection process in $\text{Suture}_{\text{RP}}$.

2) New Bugs: As mentioned earlier, we found 4 more cases that are real bugs, which previously classified as false positive by human inspection in Suture. Two bugs are from the `sound` subsystem, reported to the maintainers, while waiting for their feedback. One of them involves a data constraint that appears to sanitize a tainted value but can be bypassed due to subtle control-flow logic.

The other two bugs are from the `i2c` subsystem. They involve a condition where two tainted values must simultaneously satisfy specific constraints—a case that standard taint analyses typically miss due to their focus on single-source propagation.

We have reported all cases following responsible disclosure. Full technical details will be made available after the issues are resolved.

3) Analysis of FPs: Despite the general effectiveness of BUGLENS, it shows 10 FPs. Upon careful examination of these cases, we attribute the inaccuracies to several distinct factors:

- *Static Analysis Fundamental Limitations* (5 cases): False positives arising from inherent limitations in the underlying static analyzers that BUGLENS is not designed to address. These include imprecisions in taint tracking through complex data structures, incorrect indirect call resolution, *etc.* BUGLENS intentionally operates on the dataflow provided by the static analyzers rather than attempting to verify the accuracy of this information itself.
- *Environment and Language Understanding* (4 cases): Imprecision resulting from the LLM's incomplete grasp of C language semantics, hardware-level interactions, and kernel-specific programming patterns.

- *Internal Modeling Errors* (1 case): Inaccuracy originating from a faulty prediction by LLM.

4) Analysis of FNs: Despite the number of FN shown in the table I is 0, BUGLENS still produced several FNs but gets mitigated by majority voting. Specifically, we observed that the FNs were concentrated in the Constraint Assessor (ConA) component, and the SecIA component typically does not generate FNs due to the *arbitrary control hypothesis* (AC-Hypo), which will be discussed in §VI-C2.

We identified two primary reasons for these FNs:

- **Overlooked Complex Conditions:** The LLM sometimes failed to recognize complex conditions of the program and therefore summarize the pre-/postcondition incorrectly.
- **Misinterpreting Validation vs. Sanitization:** LLMs occasionally misclassified sanitization as validations, thereby missing vulnerabilities.

These specific failure patterns observed within ConA were the direct motivation for designing the Structured Analysis Guidance (SAG). The SAG mechanism enhances the prompts used specifically within the ConA stage, providing targeted instructions aimed at guiding the LLM to avoid these identified pitfalls (*e.g.,* explicitly probing for bypass logic, carefully differentiating data constraint types).

The positive impact of SAG in mitigating these FNs is empirically demonstrated in RQ2 (§VI-C). As shown in Table II, the Full Design configuration (using ConA with SAG prompts) consistently yields fewer FNs.

Nevertheless, the impact of SAG is not absolute. The ability of a language model to follow such complex instructions can vary significantly, particularly across different models. Our results in RQ2/RQ3 (Table II) indicate that some models could still cause some residual FNs even with the SAG.

> ***Takeaway 1.*** BUGLENS *can effectively post-refine the results of exisiting static analyzers, hugely improving the precision, and can even find missed bugs.*

Table II. Bug Analysis Performance Comparison Across LLMs and Design Approaches (Total Cases=120, Real Bugs=22)

Model	Full Design			w/o SAG			Simple Prompt		
	FN	FP	F_1	FN	FP	F_1	FN	FP	F_1
OpenAI o3-mini	0	3	**0.94**	10	1	0.67	18	7	0.24
OpenAI o1	3	6	0.81	8	6	0.67	18	6	0.25
OpenAI GPT-4.1	1	9	0.81	7	7	0.68	3	23	0.59
Gemini 2.5 Pro	12	3	0.57	14	4	0.47	6	24	0.52
Claude 3.7 Sonnet	13	2	0.54	17	2	0.34	1	51	0.44
DeepSeek R1	4	7	**0.77**	10	6	0.60	5	42	0.42

C. RQ2: Component Contribution

To address RQ2, we conduct an incremental analysis. This study evaluates the contribution of the Security Impact Assessor (SecIA), the subsequent Constraint Assessor (ConA), and the specialized Structured Analysis Guidance (SAG) design used within ConA, by comparing performance across progressively enhanced configurations of BUGLENS.

Table III. Performance of SecIA, with and without the Aribitary Control Hypothesis (AC-Hypo).

Model	w/o AC-Hypo				w/ AC-Hypo			
	FP	FN	Prec	Rec	FP	FN	Prec	Rec
OpenAI o3-mini	13	5	0.57	0.77	38	0	0.37	1.0
OpenAI o1	8	4	0.69	0.82	39	0	0.36	1.0
OpenAI GPT-4.1	15	2	0.57	0.91	36	1	0.69	0.95
Gemini 2.5 Pro	60	3	0.24	0.86	73	0	0.23	1.0
Claude 3.7 Sonnet	20	2	0.50	0.91	31	0	0.42	1.0
DeepSeek R1	25	12	0.29	0.45	71	4	0.18	0.82

We assess the performance under the following configurations:

- **Baseline**: This configuration employs the simple prompt design, which is a straightforward prompting approach without any of the specialized components (*i.e.,* directly asking *"Is this case from static analysis report vulnerable?"*). We describe this design in §III-C. The baseline design shows the performance of the task based solely on the LLM's inherent capabilities with minimal guidance, which provides a starting point for comparison.
- **+ SecIA**: Adds the SecIA component to the Baseline. *Purpose:* Comparing this to the Baseline isolates the contribution of the SecIA stage. Detailed metrics for SecIA's filtering rate and soundness are in Table III.
- **+ SecIA + ConA (w/o SAG)**: Adds the ConA component to the "+ SecIA" configuration, utilizing a simpler prompt design for constraint checking (*i.e.,* without SAG). Comparing this to "+ SecIA" isolates the contribution of adding the constraint assesses step itself.
- **Full Design (+ SecIA + ConA + SAG)**: This configuration enhances the ConA component from the previous step by incorporating the specialized SAG design. This represents the complete BUGLENS system. Comparing this to previous configurations emphasize the contribution of the SAG.

For this component analysis (RQ2) and the subsequent model versatility analysis (RQ3), we focus our evaluation on a dataset derived from Linux kernel analysis, specifically targeting the `sound` module. This module was selected because the original Suture study identified it as containing a high density of true positive vulnerabilities (22 out of 24 known bugs), providing a rich testbed for assessing bug detection capabilities. The dataset consists of 120 cases, with 22 known bugs (positives) and 98 non-bug cases (negatives).

We evaluate the performance of these components for diverse LLMs, including OpenAI's o3-mini, o1, GPT-4.1, Gemini 2.5 Pro, Claude 3.7 Sonnet, and DeepSeek R1. The overall performance results for these configurations are summarized in Table II, while Table III provides the detailed breakdown specifically for the SecIA component's effectiveness and filtering metrics.

1) Baseline: Our experimental results clearly demonstrate the significant contribution of our proposed multi-phase workflow and its components compared to a baseline approach. As shown in Table II, despite some models like Claude 3.7 Sonnet (FN=1) and GPT-4.1 (FN=3) showed low False

Negatives, potentially reflecting their raw analytical power, this came at the cost of high False Positives (FP=51 and FP=23, respectively), rendering this simple design ineffective for practical use. The F1 scores for the baseline were generally low across models. This direct prompting approach demonstrated worse performance when compared to other BUGLENS configurations.

2) Security Impact Assessor (SecIA): As Table III shows, our Arbitrary Control Hypothesis (AC-Hypo) enhances recall across all models. Without AC-Hypo, the models exhibit noticeable False Negatives, ranging from 2 to 12 FN cases across tested models. After applying AC-Hypo, the FN rate decreases to zero for all models except DeepSeek R1 and GPT-4.1, which maintains a low FN rate (4 and 1, respectively), achieving a high recall of 0.82 and 0.95.

Meanwhile, SecIA demonstrates strong effectiveness as a security vulnerability *filter*. Taking OpenAI's o3-mini as an example, among a total of 98 negative cases, SecIA successfully filters out 60 cases (TN). This indicates that SecIA not only has a high recall rate, but it is also effective, substantially improving analysis efficiency.

3) Constraint Assessor (ConA) without Structured Analysis Guidance (SAG): While this multi-phase workflow (*i.e.,* SecIA + ConA) significantly reduces the high volume of FPs seen in the Baseline; for instance, Claude 3.7 Sonnet's FPs dropped from 51 to 2, and Gemini 1.5 Pro's from 24 to 4. It also leads to a significant increase in False Negatives (FNs) for Gemini 2.5 Pro (FN=14), Claude 3.7 Sonnet and DeepSeek R1 (FN=17), and OpenAI o1 (FN=10) in the 'w/o SAG'. This supports our hypothesis (§III-C) that providing constraints, while helpful for pruning obvious non-bugs, can encourage LLMs to become overly confident. Once patterns suggesting data validity are identified, the LLM may default to classifying the issue as "not a bug," reflecting a potential statistical bias towards common safe patterns rather than performing nuanced reasoning about subtle flaws or bypass conditions.

4) Structured Analysis Guidance (SAG): Comparing the Full Design (using SAG within ConA) to the w/o SAG configuration in Table II demonstrates SAG's effectiveness. Introducing SAG leads to a substantial reduction in FNs across all tested models. Consequently, the overall F_1 score sees a marked improvement with SAG (e.g., improving from 0.67 to 0.94 for o3-mini and 0.34 to 0.54 for Claude). This indicates that SAG successfully guides the LLM within ConA to overcome the previously observed overconfidence, achieving a better balance between FP reduction and FN mitigation.

> ***Takeaway 2.*** *The design components of* BUGLENS *enables effective LLM bug analysis by significantly reducing both FP and FN compared to baseline prompting.*

D. RQ3: Model Versatility

The results shown in Table II affirm that BUGLENS is a general LLM-based technique applicable across different models, consistently improving upon baseline performance. However, the degree of success highlights variations in how different LLMs interact with complex instructions and structured reasoning processes.

As noted in RQ2, the baseline performance offers a glimpse into the models' raw capabilities, somewhat correlating with general LLM benchmarks where Gemini 2.5 Pro and Claude 3.7 Sonnet are often considered leaders [30]. However, this raw capability did not directly translate to superior performance within our structured task without significant guidance.

When employing the 'Full Design' of BUGLENS, we observed distinct differences in instruction-following adherence. The OpenAI models, o1, GPT-4.1 (F1=0.81), and particularly our core model o3-mini (F1=0.94), demonstrated excellent alignment with the workflow's intent.

Conversely, while the 'Full Design' significantly improved the F1 scores for Gemini 2.5 Pro (0.57) and Claude 3.7 Sonnet (0.54) compared to their baseline or 'w/o sag' results by drastically cutting down FPs, they still struggled with relatively high false negatives (FN=12 and FN=13, respectively). This suggests that even with the SAG, these powerful models may face challenges in precisely balancing the various analytical steps or interpreting the nuanced instructions within our workflow, possibly still exhibiting a degree of the over-confidence (for "sanity check") that SAG could not fully overcome in their case. DeepSeek R1 (F1=0.77) showed a strong, balanced improvement, landing between the GPT models and the Gemini/Claude in terms of final performance with the full design. This demonstrates the generality of the BUGLENS to open-sourced models.

In summary, while our approach is broadly applicable, its optimal performance depends on the LLM's ability to robustly follow complex, multi-step instructions, with models like OpenAI's o3-mini currently showing the strongest capability in this specific structured bug analysis task.

> ***Takeaway 3.*** BUGLENS *shows broad applicability and yields promising results across diverse LLMs, including the open-source DeepSeek R1. OpenAI's o3-mini currently gets the best result.*

E. Case Study: Data Structure Traversal

Linux kernel code often uses pointers to traverse data structures. For example, the following code walks through a linked list `list` using a marco `list_for_each_entry`:

```
1  struct snd_kcontrol *snd_ctl_find_id(... *id ){
2  ...
3    if (id->numid != 0)
4      return snd_ctl_find_numid(card, id->numid);
5    list_for_each_entry(kctl, ...){
6      ...
7     if (kctl->id.index > id->index )
8      continue;
9     if (kctl->id.index + kctl->count <= id->index )
10     continue;
11    return kctl;
12   }
13   return NULL;
14 }
```

In this code, the loop goes through each element in the list and checks if any of them match the input id, which

comes from the user. The user input (`*id`) only decides which element gets picked, not how long the loop runs. However, the static analyzer, Suture, misclassifies this case as a potential *tainted loop bound* warning. Our method, BUGLENS, avoids this false warning by using the LLM's more nuanced understanding of how data structures like linked lists work.

Tracking the returned `kctl` object at Line 11, we can find the following range constraints for the tainted value `id->index` (from Line 7 to 11):

```
/* Range constraints for id->index */
id->index >= kctl->id.index
id->index < kctl->id.index + kctl->count
```

Where the `kctl` object is returned once the loop ends (*i.e.,* `kctl != NULL`).

Notably, there's a **bypassable condition** in this function. Before the list iteration, there is a check if `id->numid` is not zero (Line 3 to 4). If it is not zero, the function will return the result of `snd_ctl_find_numid(card, id->numid)`, and the loop will not run.

This means the checks inside the loop (Line 7 to 11) for `id->index` might *not* happen at all. Therefore, the `id->index` is only validated when `id->numid` is zero (and the `kctl` object is returned).

F. Threats to Validity

Dependency on LLMs. The performance of BUGLENS is influenced by the capabilities of the underlying LLMs used for reasoning. While this dependency could potentially impact external validity, our experiments demonstrate BUGLENS's robustness across different models. We have evaluated BUGLENS with multiple LLMs, including both closed-source and open-source models like Deepseek R1, which achieves approximately 80% of the performance of top-performing models. These results confirm that BUGLENS's approach generalizes well across different models, though performance variations exist.

Limited Checkers and Bug Types. Our evaluation uses only taint-style checkers—those inherited from and Suture (and Dr.checker), plus one CodeQL port—focused on memory-safety and DoS bugs in the Linux kernel. This narrow setup threatens external validity, and the performance we report may not carry over to other bugs (*e.g.,* data races) or analysis frameworks. Although the extra CodeQL checker suggests BUGLENS can transfer across tooling, a broader study is needed for general applicability.

VII. DISCUSSION

Towards More Sound Analysis. The soundness of current implementation could be improved through two directions: (1) Adding symbolic verification [31], [32] to validate the LLM's reasoning and refine outputs based on formal methods (2) Implementing a hybrid architecture where the LLM performs initial code slicing while symbolic execution handles constraint analysis, combining the LLM's contextual understanding with provably sound formal reasoning.

Integration with More Analyzers. It is a worthwhile goal to explore how LLMs can complement state-of-the-art static analysis tools for complex programs such as the Linux kernel [33]–[39], which often make tradeoffs to sacrifice precision for scalability. Similarly, CodeQL-based detectors (beyond CodeQL-OOB) are low-precision in nature and can benefit from solutions like ours. Compared to heavier static analysis tools such as Suture, an imprecise static analysis could be much easier to implement and maintain.

VIII. RELATED WORK

LLM for Program Analyses & Bug Detection LLMs have been widely applied to program analysis tasks for bug detection. IRIS [19] combines LLMs' contextual understanding with CodeQL queries to enhance taint analysis. LLM4SA [23] leverages LLMs to refine static analysis results. Their approach serves as a baseline for BUGLENS (referred as simple prompt). LLMDFA and LLMSAN [20], [40] use LLMs to perform data flow analysis and bug detection. LLift [21] focuses on use-before-intialization bugs and prompts LLMs to identify and summarize possible initializers. Focusing on real-world problems, BUGLENS upgrades the scope to a general taint-style bug detection cross multiple functions.

Reasoning for LLMs. Despite their success on many tasks, the ability of LLMs to reason about code semantics and behaviors remains an active area of research [41]–[44]. Recent studies have shown that LLMs are still far from performing reliable code reasoning, and their predictions are thus fragile and susceptible to superficial changes in input [45]–[47]. This fragility is often attributed to the learned models taking "shortcuts" based on superficial patterns in training data rather than robust, generalizable reasoning strategies [24], [48]–[51]. BUGLENS mitigates this problem with boosting the LLMs' reasoning by constraining their reasoning space with structural and symbolic procedures [52]–[56].

IX. CONCLUSION

This paper introduces BUGLENS, an innovative post-refinement framework that integrates Large Language Models (LLMs) with static analysis. By employing Security Impact Assessor (SecIA), Constraint Assessor (ConA), and Structured Analysis Guidance (SAG) to guide LLMs through the reasoning process, BUGLENS significantly enhances the precision of initial static analysis findings without sacrificing scalability. Our evaluation demonstrates that BUGLENS dramatically reduces false positives in Linux kernel Analysis, minimizes manual inspection effort, and uncovers previously ignored vulnerabilities, highlighting the promise of guided LLMs in making automated bug detection more practical and effective.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and valuable suggestions. This material is based upon work supported by the United States Air Force and DARPA under Agreement No. FA8750-24-2-0002.

References

[1] A. Gosain and G. Sharma, "Static Analysis: A Survey of Techniques and Tools," in *Intelligent Computing and Applications*, ser. Advances in Intelligent Systems and Computing, D. Mandal, R. Kar, S. Das, and B. K. Panigrahi, Eds. New Delhi: Springer India, 2015, pp. 581–591.

[2] J. Park, H. Lee, and S. Ryu, "A survey of parametric static analysis," *ACM Comput. Surv.*, vol. 54, no. 7, pp. 149:1–149:37, 2022. [Online]. Available: https://doi.org/10.1145/3464457

[3] G. Horvath, R. Kovacs, and Z. Porkolab, "Scaling Symbolic Execution to Large Software Systems," Aug. 2024, arXiv:2408.01909 [cs]. [Online]. Available: http://arxiv.org/abs/2408.01909

[4] Y. Zhai, Y. Hao, H. Zhang, D. Wang, C. Song, Z. Qian, M. Lesani, S. V. Krishnamurthy, and P. Yu, "Ubitect: A precise and scalable method to detect use-before-initialization bugs in linux kernel," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, 2020.

[5] H. Zhang, W. Chen, Y. Hao, G. Li, Y. Zhai, X. Zou, and Z. Qian, "Statically Discovering High-Order Taint Style Vulnerabilities in OS Kernels," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Virtual Event Republic of Korea: ACM, Nov. 2021, pp. 811–824. [Online]. Available: https://dl.acm.org/doi/10.1145/3460120.3484798

[6] A. Khare, S. Dutta, Z. Li, A. Solko-Breslin, R. Alur, and M. Naik, "Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities," Oct. 2024, arXiv:2311.16169 [cs]. [Online]. Available: http://arxiv.org/abs/2311.16169

[7] B. A. Stoica, U. Sethi, Y. Su, C. Zhou, S. Lu, J. Mace, M. Musuvathi, and S. Nath, "If At First You Don't Succeed, Try, Try, Again...? Insights and LLM-informed Tooling for Detecting Retry Bugs in Software Systems," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. Austin TX USA: ACM, Nov. 2024, pp. 63–78. [Online]. Available: https://dl.acm.org/doi/10.1145/3694715.3695971

[8] C. Fang, N. Miao, S. Srivastav, J. Liu, N. Nazari, and H. Homayoun, "Large Language Models for Code Analysis: Do LLMs Really Do Their Job?" in *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Aug. 2024.

[9] X. Qian, X. Zheng, Y. He, S. Yang, and L. Cavallaro, "LAMD: Context-driven Android Malware Detection and Classification with LLMs," *arXiv preprint arXiv:2502.13055*, 2025. [Online]. Available: https://arxiv.org/abs/2502.13055

[10] H. He, X. Lin, Z. Weng, R. Zhao, S. Gan, L. Chen, Y. Ji, J. Wang, and Z. Xue, "Code is not Natural Language: Unlock the Power of Semantics-Oriented Graph Representation for Binary Code Similarity Detection," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 1759–1776. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/he-haojie

[11] S. Kambhampati, "Can large language models reason and plan?" *Annals of the New York Academy of Sciences*, vol. 1534, no. 1, pp. 15–18, Mar. 2024. [Online]. Available: https://doi.org/10.1111/nyas.15125

[12] J. Chen, Z. Pan, X. Hu, Z. Li, G. Li, and X. Xia, "Reasoning runtime behavior of a program with llm: How far are we?" in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, 2025.

[13] P. J. Chapman, C. Rubio-González, and A. V. Thakur, "Interleaving static analysis and LLM prompting with applications to error specification inference," *International Journal on Software Tools for Technology Transfer*, Feb. 2025. [Online]. Available: https://doi.org/10.1007/s10009-025-00780-7

[14] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "{DR}. {CHECKER}: A Soundy Analysis for Linux Kernel Drivers," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1007–1024.

[15] GitHub, "CodeQL: The libraries and queries that power security researchers around the world," https://codeql.github.com/, 2025, accessed: 2025-05-29.

[16] Github (2025), "About data flow analysis — CodeQL," 2025. [Online]. Available: https://codeql.github.com/docs/writing-codeql-queries/about-data-flow-analysis/

[17] K. Backhouse, "Stack buffer overflow in Qualcomm MSM 4.4 - Finding bugs with CodeQL," Jan. 2018. [Online]. Available: https://securitylab.github.com/research/stack-buffer-overflow-qualcomm-msm/

[18] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, M. Luca, P. O'Hearn, I. Papakonstantinou, and D. Rodriguez, "Moving Fast with Software Verification," 2015. [Online]. Available: https://research.facebook.com/publications/moving-fast-with-software-verification/

[19] Z. Li, S. Dutta, and M. Naik, "IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities," in *The Thirteenth International Conference on Learning Representations (ICLR 2025)*, 2025. [Online]. Available: http://arxiv.org/abs/2405.17238

[20] C. Wang, W. Zhang, Z. Su, X. Xu, X. Xie, and X. Zhang, "LLMDFA: analyzing dataflow in code with large language models," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024. [Online]. Available: http://papers.nips.cc/paper_files/paper/2024/hash/ed9dcde1eb9c597f68c1d375bbecf3fc-Abstract-Conference.html

[21] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages (PACMPL), Volume 8, Issue OOPSLA1*, vol. 8, no. OOPSLA1, 2024.

[22] M. Zheng, D. Xie, Q. Shi, C. Wang, and X. Zhang, "Validating Network Protocol Parsers with Traceable RFC Document Interpretation," in *Proceedings of the 2025 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2025, Trondheim, Norway, 2025.

[23] 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 Trans. Knowl. Discov. Data*, vol. 18, no. 7, pp. 168:1–168:34, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3653718

[24] R. T. McCoy, S. Yao, D. Friedman, M. D. Hardy, and T. L. Griffiths, "Embers of autoregression show how large language models are shaped by the problem they are trained to solve," *Proceedings of the National Academy of Sciences*, vol. 121, no. 41, p. e2322420121, 2024. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.2322420121

[25] A. Prabhakar, T. L. Griffiths, and R. T. McCoy, "Deciphering the Factors Influencing the Efficacy of Chain-of-Thought: Probability, Memorization, and Noisy Reasoning," Oct. 2024, arXiv:2407.01687 [cs]. [Online]. Available: http://arxiv.org/abs/2407.01687

[26] C. Snell, J. Lee, K. Xu, and A. Kumar, "Scaling llm test-time compute optimally can be more effective than scaling model parameters," *arXiv preprint arXiv:2408.03314*, 2024.

[27] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," Jan. 2023, arXiv:2201.11903 [cs]. [Online]. Available: http://arxiv.org/abs/2201.11903

[28] R. Kopathy, "ruben2020/codequery," Mar. 2025. [Online]. Available: https://github.com/ruben2020/codequery

[29] GitHub, "Codeql," https://codeql.github.com, 2025.

[30] V. AI, "LLM Leaderboard," https://www.vellum.ai/llm-leaderboard, 2025, accessed: 2025-04-15.

[31] S. Bhatia, J. Qiu, N. Hasabnis, S. A. Seshia, and A. Cheung, "Verified Code Transpilation with LLMs," *38th Conference on Neural Information Processing Systems (NeurIPS 2024)*, 2024.

[32] Y. Cai, Z. Hou, D. Sanan, X. Luan, Y. Lin, J. Sun, and J. S. Dong, "Automated Program Refinement: Guide and Verify Code Large Language Model with Refinement Calculus," *Proc. ACM Program. Lang.*, vol. 9, no. POPL, pp. 69:2057–69:2089, Jan. 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3704905

[33] G. Li, H. Zhang, J. Zhou, W. Shen, Y. Sui, and Z. Qian, "A hybrid alias analysis and its application to global variable protection in the linux kernel," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 4211–4228. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/li-guoren

[34] G. Li, M. Sridharan, and Z. Qian, "Redefining indirect call analysis with kallgraph," in *IEEE Security and Privacy*, 2025.

[35] H. Zhang, J. Kim, C. Yuan, Z. Qian, and T. Kim, "Statically discover cross-entry use-after-free vulnerabilities in the linux kernel," in *32nd Annual Network and Distributed System Security Symposium, NDSS*, 2025.

[36] 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)*. Anaheim, CA: USENIX

Association, Aug. 2023, pp. 3727–3744. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/cai-yuandao
[37] D. Liu, S. Ji, K. Lu, and Q. He, "Improving indirect-call analysis in llvm with type and data-flow co-analysis," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.
[38] J. Liu, L. Yi, W. Chen, C. Song, Z. Qian, and Q. Yi, "LinKRID: Vetting imbalance reference counting in linux kernel with symbolic execution," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 125–142. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/liu-jian
[39] Y. Zhai, Y. Hao, Z. Zhang, W. Chen, G. Li, Z. Qian, C. Song, M. Sridharan, S. V. Krishnamurthy, T. Jaeger, and P. Yu, "Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel," in *Proceedings of the 2020 ISOC Network and Distributed Systems Security Symposium (NDSS)*, Feb. 2022.
[40] C. Wang, W. Zhang, Z. Su, X. Xu, and X. Zhang, "Sanitizing large language models in bug detection with data-flow," in *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*, Y. Al-Onaizan, M. Bansal, and Y. Chen, Eds. Association for Computational Linguistics, 2024, pp. 3790–3805. [Online]. Available: https://aclanthology.org/2024.findings-emnlp.217
[41] Y. Ding, J. Peng, M. J. Min, G. Kaiser, J. Yang, and B. Ray, "Semcoder: Training code language models with comprehensive semantics," *arXiv preprint arXiv:2406.01006*, 2024.
[42] E. Zelikman, Y. Wu, J. Mu, and N. Goodman, "Star: Bootstrapping reasoning with reasoning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 15 476–15 488, 2022.
[43] J. Li, D. Guo, D. Yang, R. Xu, Y. Wu, and J. He, "Codei/o: Condensing reasoning patterns via code input-output prediction," *arXiv preprint arXiv:2502.07316*, 2025.
[44] A. Ni, M. Allamanis, A. Cohan, Y. Deng, K. Shi, C. Sutton, and P. Yin, "Next: Teaching large language models to reason about code execution," *arXiv preprint arXiv:2404.14662*, 2024.
[45] K. Pei, W. Li, Q. Jin, S. Liu, S. Geng, L. Cavallaro, J. Yang, and S. Jana, "Exploiting code symmetries for learning program semantics," *arXiv preprint arXiv:2308.03312*, 2023.
[46] B. Steenhoek, M. M. Rahman, M. K. Roy, M. S. Alam, H. Tong, S. Das, E. T. Barr, and W. Le, "To Err is Machine: Vulnerability Detection Challenges LLM Reasoning," Jan. 2025, arXiv:2403.17218 [cs]. [Online]. Available: http://arxiv.org/abs/2403.17218
[47] A. Hochlehnert, H. Bhatnagar, V. Udandarao, S. Albanie, A. Prabhu, and M. Bethge, "A sober look at progress in language model reasoning: Pitfalls and paths to reproducibility," *arXiv preprint arXiv:2504.07086*, 2025.
[48] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.
[49] F. Gao, Y. Wang, and K. Wang, "Discrete adversarial attack to models of code," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 172–195, 2023.
[50] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1482–1493.
[51] P. Bielik and M. Vechev, "Adversarial robustness for code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 896–907.
[52] J. Li, G. Li, Y. Li, and Z. Jin, "Structured chain-of-thought prompting for code generation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1–23, 2025.
[53] W. Chen, X. Ma, X. Wang, and W. W. Cohen, "Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks," *arXiv preprint arXiv:2211.12588*, 2022.
[54] C. Li, J. Liang, A. Zeng, X. Chen, K. Hausman, D. Sadigh, S. Levine, L. Fei-Fei, F. Xia, and B. Ichter, "Chain of code: Reasoning with a language model-augmented code emulator," *arXiv preprint arXiv:2312.04474*, 2023.
[55] Y. Chen, H. Jhamtani, S. Sharma, C. Fan, and C. Wang, "Steering large language models between code execution and textual reasoning," *arXiv preprint arXiv:2410.03524*, 2024.
[56] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," *Advances in neural information processing systems*, vol. 36, pp. 11 809–11 822, 2023.