Exploring Static Taint Analysis in LLMs: A Dynamic Benchmarking Framework for Measurement and Enhancement

Haoran Zhao[1], Lei Zhang[2], Keke Lian[2], Fute Sun[1], Bofei Chen[1],
Yongheng Liu[1], Zhiyu Wu[1], Yuan Zhang[2], and Min Yang[2]
1: Fudan University, {zhaohr23, ftsun22, bfchen22, yhliu24, wuzy24}@m.fudan.edu.cn,
2: Fudan University, {zxl, kklian20, yuanxzhang, m_yang}@fudan.edu.cn

***Abstract*—LLMs offer a promising avenue to overcome the limitations of traditional taint analysis techniques, with a growing number of studies leveraging LLMs for taint analysis and its downstream applications. However, these studies lack a systematic understanding of LLMs' taint analysis capabilities, limiting their transferability and reliability. To bridge this gap and better apply LLMs to static taint analysis, we aim to comprehensively measure and understand LLMs' taint analysis capabilities.**

Using existing benchmarks is a straightforward approach, but they are unsuitable due to issues such as training data leakage, not accounting for LLMs' features, and improper assessment criteria. Manually constructing new benchmarks is not only labor-intensive but also struggles to remain effective as LLMs evolve. To address these, we propose LLMCapLens, a dynamic benchmark generation framework to systematically measure and enhance LLMs' capabilities. LLMCapLens models influencing factors of LLMs' taint analysis capabilities, employing a Basic Unit-Based generation method and a lightweight dynamic taint analysis-based verification method to implement the automated generation of targeted benchmarks, ensuring both diversity and correctness. Furthermore, LLMCapLens proposes a measurement-driven, training-free, model-specific enhancement approach.

We apply LLMCapLens to 10 mainstream LLMs, revealing how they perform under various influencing factors and identifying unique characteristics, such as the underlying error causes for each model. Notably, our enhancement approach significantly improves LLM performance—GPT-4 Turbo, for instance, achieved improvements across 16 out of 19 factors, with an average True Negative Rate increase of 21.29%. Finally, we validate the real-world impact of our method by applying enhanced LLMs to vulnerability detection, demonstrating a substantial improvement over prior approaches.

I. Introduction

Large Language Models (LLMs), such as GPT-4, have rapidly advanced in recent years. Beyond their prowess in natural language processing (NLP), LLMs also exhibit strong performance in various code-related tasks. Consequently, there is an increasing body of research [1], [2], [3], [4], [5], [6] exploring the application of LLMs in program analysis tasks.

Static taint analysis is a crucial technique widely used in program analysis tasks such as vulnerability detection. Traditional static taint analysis has inherent limitations, such as limited semantic understanding, while LLMs offering a promising avenue for overcoming these challenges. Hence, more and more studies are applying LLMs to taint analysis. However, these works [7], [1], [2], [3], [4] always treat LLMs as black boxes, feeding them code snippets along with high-level prompts like 'Can user input taint the parameters of this function?'. Such an application lacks both explanations for LLMs' behavior and evaluations of their performance across different contexts, e.g., different code structures, limiting their transferability and reliability. In fact, existing studies have drawn contradictory conclusions, with some claiming LLMs are ready for such tasks [2], [4], [3], while others argue they are not [7], [8]. This disparity highlights the risks of using LLMs without a thorough understanding of their taint analysis capabilities. A systematic measurement of their strengths and limitations is essential for their effective integration into program analysis tasks.

Directly using existing taint analysis benchmarks to measure LLMs' capabilities is a straightforward approach. However, they are not suitable due to the following reasons:

- First, existing benchmarks and real-world code may overlap with LLMs' training data [7], making them unreliable for accurately measuring LLMs' capabilities.
- Second, traditional benchmarks assess taint analysis via vulnerability detection [9], [10], but LLMs may reach answers using alternative strategies, such as pattern matching, rather than actual taint analysis. This makes it difficult to determine whether an LLM is truly performing taint tracking.
- Third, these benchmarks do not account for LLM-specific factors, such as sensitivity to code semantics and attention mechanisms. Unlike traditional analysis tools, LLMs' responses can be influenced by superficial variations, such as code length or comments, even when the underlying taint-related issues remain the same. Existing benchmarks fail to distinguish such effects.

Therefore, a new benchmark is needed to comprehensively measure LLM's taint analysis capability. However, manually constructing such a benchmark is highly labor-intensive and lacks scalability, making it hard to remain effective as LLMs evolve or face specific requirements. For instance, the rise of deep reasoning LLMs necessitates test cases with higher reasoning complexity, while evaluating LLMs' semantic understanding requires more code-semantic-related cases. Manu-

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00082

ally creating benchmarks to satisfy all possible requirements is unrealistic, especially given the rapid advancement of LLMs. To address this, we propose a dynamic benchmark generation approach that automatically produces benchmarks tailored to specific needs. Compared with static benchmarks, our approach enables more comprehensive and targeted measurement while significantly reducing manual effort and offering great scalability. However, this presents challenges:

- Identify influencing factors. Identifying influencing factors of LLMs' taint analysis capabilities is fundamental to ensuring the diversity of the generated benchmarks. However, this is not an easy task as traditional benchmarks often test by aggregating samples without systematically modeling influencing factors. Besides, we should account for LLM-specific factors, such as semantic elements and code length.
- Balancing test cases' correctness and complexity. The approach must be capable of generating sufficiently complex test cases(e.g., multi-level nesting) while ensuring their correctness, including both syntactic and semantic correctness.
- Automatic answer verification. Besides constructing test cases, manually labeling each case is also labor-intensive. Therefore, an automated way to obtain expected answers and compare them with LLM outputs is needed. This is not an easy task, as existing taint analysis tools have limitations when dealing with complex test cases.

We address these challenges by proposing LLMCAPLENS, a dynamic benchmark generation framework designed to systematically measure and enhance LLMs' static taint analysis capabilities. LLMCAPLENS consists of four modules: *LLM Taint Analysis Influencing Factors Modeling*, *Test Case Automatic Generation*, *Test Case Automatic Verification*, and *Measurement-Driven Enhancement*. The first three modules dynamically generate benchmarks and measure LLMs' capabilities, while the final module leverages the measurement results to enhance LLMs' taint analysis capabilities.

Specifically, *LLM Taint Analysis Influencing Factors Modeling* identifies 19 code structural factors and three semantic factors by analyzing two benchmarks and 16 academic papers, ensuring the diversity of generated benchmarks. In *Test Case Automatic Generation*, LLMCAPLENS employs a Basic Unit-based method to dynamically generate test cases with minimal manual effort while maintaining correctness and complexity. In *Test Case Automatic Verification*, LLMCAPLENS decomposes complex cases into simple units and applies a lightweight dynamic taint analysis to automatically compute ground truth, reducing manual labeling effort and errors. Finally, *Measurement-Driven Enhancement* leverages measurement results to improve LLMs' taint analysis capabilities, unlike approaches that optimize for specific cases, LLMCAPLENS focuses on enhancing LLMs' inherent abilities in a model-specific manner.

We designed three measurement plans—single-factor, multi-factor, and semantic-factor—and leveraged LLMCAPLENS to generate benchmarks, evaluating the taint analysis capabilities of 10 mainstream LLMs. Our experiments yielded several key findings. For example, we observed that LLMs often exhibit various biases, such as a tendency to provide positive responses, resulting in high True Positive Rates (TPR) and low True Negative Rates (TNR); Certain code structures, such as loop structures, consistently challenge LLMs across models, and code comments significantly influence LLMs' judgments, highlighting their sensitivity to semantic cues. These findings provide valuable insights into LLMs' strengths and limitations in taint analysis. For instance, they suggest avoiding programs with extensive loop structures when using LLMs for analysis and leveraging additional comments to improve performance. Moreover, our findings can facilitate better collaboration between LLMs and traditional analysis tools, as discussed in §V.

The Measurement-Driven Enhancement method has proven highly effective across multiple LLMs. For example, GPT-4 Turbo showed accuracy improvements in 16 out of 19 factors, with an average TNR increase of 21.29%. To demonstrate the practical impact of our approach, we further evaluated the enhanced LLMs on vulnerability detection tasks. Using vulnerable code and corresponding patched versions from NIST SARD[9], we compared our method against prior work[7]. As a result, our approach enabled LLMs to detect subtle code modifications and their effects on vulnerabilities, resulting in significantly higher accuracy in vulnerability detection.

We summarize the contributions of this paper as follows:

- We develop LLMCAPLENS, a dynamic benchmark generation framework designed to systematically measure and enhance LLMs' capabilities in static taint analysis. It models 19 code structural and 3 code semantic influencing factors, enabling automated test case generation and verification. Additionally, it contains a Measurement-Driven Enhancement method to enhance LLMs' taint analysis capability.
- We designed three measurement plans and leveraged LLMCAPLENS to measure 10 widely used LLMs, gaining comprehensive insights into their taint analysis capabilities, and demonstrating the effectiveness of our enhancement method in vulnerability detection tasks.
- We develop a new benchmark using LLMCAPLENS that provides more granular insights for LLMs than traditional benchmarks. Moreover, LLMCAPLENS can be easily extended to generate other benchmarks, such as more targeted benchmarks for traditional taint analysis tools.

II. LLMCAPLENS DESIGN

Figure 1 presents an overview of LLMCAPLENS. LLMCAPLENS consists of four main components: Influencing Factor Modeling (§II-A), Automated Test Case Generation (§II-B), Automated Test Case Verification(§II-C) and Measurement-Driven Enhancement (§II-D). The Influencing Factor Modeling component identifies and models key influencing factors of LLM's taint analysis capability. The Automated Test Case Generation and Verification component dynamically generates benchmarks using these factors and verifies their answers. Finally, the Measurement-Driven Enhancement component leverages measurement results and the self-correction mechanism to enhance the LLMs' capability.

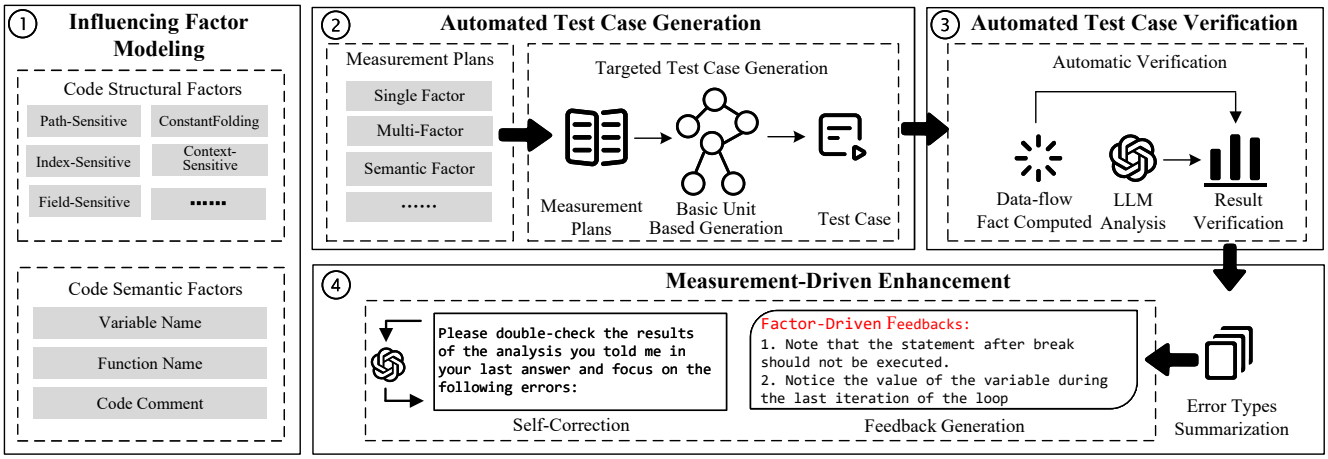

Fig. 1: The Overview Architecture of LLMCAPLENS.

```
List<String> VARNAME_1 = new ArrayList<String>();
VARNAME_1.add("safe", VARNAME_3, "moresafe" );
VARNAME_1.remove(0);
VARNAME_2 = VARNAME_1.get(0)
```
List-Index-Sensitive Unit1

```
List<String> VARNAME_1 = new ArrayList<String>();
VARNAME_1.add("safe", "moresafe", VARNAME_3);
VARNAME_1.remove(0);
VARNAME_2 = VARNAME_1.get(0)
```
List-Index-Sensitive Unit2

```
for(int VARNAME_1=0;VARNAME_1 += 1;VARNAME_1 < 70)
{...}
```
For-Loop1 Unit1

```
for(int VARNAME_1=0;VARNAME_1 += 1;VARNAME_1 < 70)
{...break;...}
```
For-Loop1 Unit2

Fig. 2: A subset of Basic Units

TABLE I: Code Structural Factors

Factor	Description	#Unit
	Control Flow-Related	
For-Loop$_1$	The number of iterations in for-loop affects the taint status.	10
For-Loop$_2$	The result of for-loop affects the taint status.	10
If-Branch	The condition of if-branch affects the taint status.	10
Sequential-Sensitive	The execution order affects the taint status.	4
Switch-Branch	The condition of switch-branch affects the taint status.	6
While-Loop$_1$	The number of iterations in while-loop affects the taint status.	10
While-Loop$_2$	The result of while-loop affects the taint status.	10
If-Path-Sensitive	The two branches of an if-branch result in different taint states.	6
Switch-Path-Sensitive	The different branches of a switch-branch result in different taint states.	6
	Data Flow-Related	
Const-Folding	Inferable constants affect the taint status.	6
String-Branch	Certain characters within a string affect the taint status.	4
Context-Sensitive	Distinguish between different calling contexts.	4
Field-Sensitive	Different fields within a class differently affect the taint status.	4
Static Propagation	Static fields or global variables affect the taint state.	4
List-Index-Sensitive	Different elements in a list differently affect the taint status.	6
Map-Index-Sensitive	Different elements in a map differently affect the taint status.	4
Object-Sensitive	Different objects of the same class differently affect the taint status.	4
Alias-Sensitive	Variable aliases(e.g., multiple references to the same object) affect the taint state.	4
Invoke-Sensitive	Multiple potential calls (e.g., call an interface method) differently affect the taint status.	6

A. *Influencing Factor Modeling*

Identifying influencing factors of LLMs' taint analysis capabilities is fundamental to ensuring the diversity of the generated benchmarks. Considering LLMs' taint analysis is shaped not only by code structures but also by their text-based semantic understanding, factors are categorized into two main types: code structural factors and code semantic factors, with code structural factors further divided into control flow-related and data flow-related types.

Code Structural Factors. Code structural factors refer to syntactic elements that influence LLMs' taint analysis capabilities, including factors affecting complexity (e.g., conditional and loop structures) and sensitivity (e.g., field and context sensitivity). While prior researches consider these factors, they often focus on specific aspects (like field sensitivity) and lack comprehensive modeling. Through analyzing two well-known benchmarks (NIST-SARD [9] and OWASP [10]) and numerous classic papers [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], we identified 19 code structural factors.

Code Semantic Factors. Some studies [8], [7], [27] have suggested that LLMs attempt to understand and utilize semantics within programs. Specifically, variable names, function names, and comments, which often carry rich semantic information[7], [8], [27], are selected as key factors to assess how code semantics influence LLMs' taint analysis capability. The details and examples are presented in §III-A.

Basic Generation Units. Factors are abstract concepts, should be mapped to concrete implementations to measure LLMs' capabilities, termed Basic Generation Units (Basic Units). To ensure representativeness when collecting units, we considered the following two aspects: (1) covering common syntax variations, such as step sizes, break statements, and initial conditions in for-loops; (2) including both positive and negative cases, such as inserting taints at various positions in `ListIndexSensitive`. Some units are shown

in Figure 2, where they either insert or delete elements at different positions, or use a 'break' statement. The number of Basic Units varies across factors (see Table I), reflecting the diversity of each factor. For-loops, for example, have many implementations (e.g., different iteration bounds, step sizes), while Sequential-Sensitive need only a few units (e.g., altering the relative position between the sanitizer and sink points). To build the unit set, we analyzed syntax variations from existing benchmarks [9], [10] and designed both positive and negative cases, following traditional benchmark construction methods [28]. For OWASP, done in one week by two security experts with 3 years' experience.

B. Automated Test Case Generation

This component takes user-defined measurement plans as input and dynamically generates benchmarks based on them, ensuring the test cases are both correct and complex.

We propose a Basic Generation Unit-based method for automatic test case generation. This method not only generates test cases but also helps automatically verify the test cases' results. (detailed in §II-C). Our key insight is to use manually crafted 'building blocks' (i.e., Basic Generation Units) to automatically generate test cases. We ensure the correctness and complexity of test cases through meticulous design.

Correctness. Correctness refers to three properties of the generated test cases: (1) syntactic validity, the test case can be compiled and executed; (2) verifiability, the test case can be automatically verified for data flow facts; and (3) semantic editability, the test case can embody diverse semantics. We ensure correctness by carefully designing the structure of Basic Units. A detailed example of Basic Units is shown in Figure 4, consisting of a code snippet and descriptive metadata. The code snippet serves as the raw material for test case generation, while the descriptive metadata guides the generation process. Specifically, the metadata includes the `[IST]`, `[OST]`, `[RULE]`, `<CTAG>`, and `[EXIT]` sections.

- `[IST]`, the Input Symbol Table, specifies which types of variables from other units the current basic unit can accept.
- `[OST]`, the Output Symbol Table, defines which variables the current basic unit can expose for use by other basic units.
- `[RULE]`, guides the dynamic taint analysis after test case generation and is used to automatically verify data flow facts, for example, for the statement `String VARNAME_1 = VARNAME_2 + VARNAME_3;`, insert the rule `[RULE] VARNAME_1.tag = VARNAME_2.tag || VARNAME_3.tag;`, as detailed in §II-C.
- `<CTAG>`, is used to add comment to the test cases.
- `[EXIT]`, this tag defines the connection points of the Basic Unit. A Basic Unit can include multiple [EXIT] tags, each requiring a corresponding [IST] and [OST].

With the help of this auxiliary information, LLMCapLens ensures the correctness of generated test cases. `[IST]`, `[OST]`, and `[EXIT]` ensure the syntactic validity; `[IST]` and `[OST]` guarantee correct variable propagation across units by recording variable types so that only variables of the same

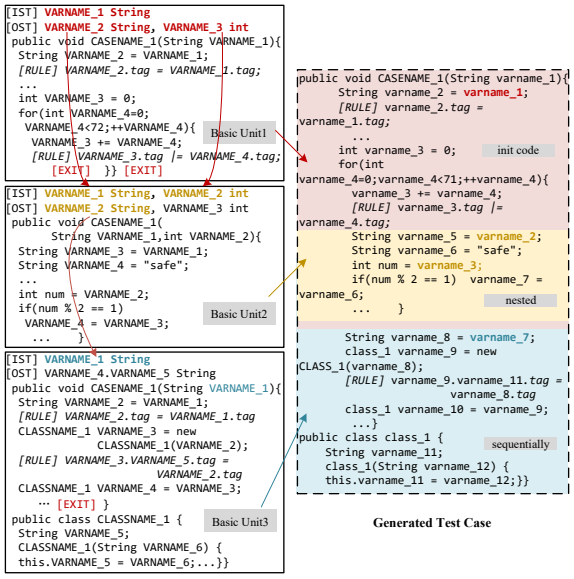

Fig. 3: An example process of generating a test case

type, or those that can be safely converted, are connected through assignments, while `[EXIT]` prevents syntax violations, such as disallowing an if statement within a loop's condition. `[RULE]` ensures verifiability by embedding taint propagation rules for automated verification in later stages. `<CTAG>` and the placeholders ensure semantic editability, allowing different semantics to be injected by retaining or removing comments and replacing variable and function names with semantically rich identifiers.

Furthermore, to ensure the determinism of taint flow facts in each test case, LLMCapLens follows the design principles of existing benchmarks [9], [10] and avoids using tainted values for control flow constraints when generating test cases. An example of generating a test case using various units is shown in Figure 3.

Complexity. Complexity of generated test cases related to three factors: (1) the complexity of individual unit x, (2) the number of units used y, and (3) the types of combined operations z. For the complexity of individual unit x, LLMCapLens contains 190 units covering common code patterns and features such as virtual calls and reflection, enabling the testing of diverse syntax and semantics. For the number of units used y, since each Basic Unit can be reused, y has no upper bound, and thus the complexity of test cases is theoretically unbounded, as increasingly complex test cases can be iteratively built from prior ones. For the types of combined operations z, although join points between units (i.e., [EXIT] tags) are restrict to ensure correctness, units can still be combined in various ways(e.g., whether nested in loops, depending on the position of [EXIT]) to produce rich scenarios; for example, nested units with conditional branches yield complex control-flow cases, while those with interprocedural calls generate dataflow cases spanning multiple

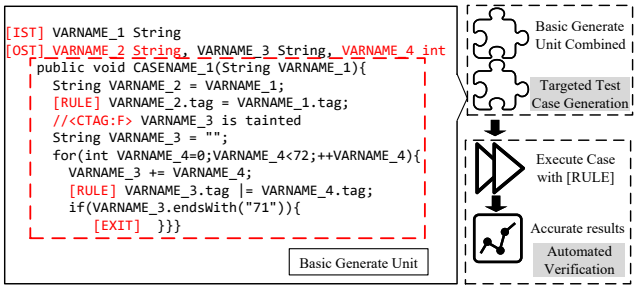

Fig. 4: Targeted Test Case Generation and Automated Verification Process

functions. This design scales exponentially, allowing a small set of Basic Units to generate a vast number of diverse test cases, as shown in §III, which can cause SOTA tools to produce numerous errors.

The entire test case generation process is divided into two phases: the Code Structure Generation phase and the Code Semantic Generation phase. In the Code Structure Generation phase, LLMCAPLENS integrates structural factors into the test case by appending units from Basic Generation Units, replacing placeholders and dividing the snippet into main code (method body of `CASENAME_XX`) and support code (auxiliary definitions for execution). LLMCAPLENS appends the main code to the [EXIT] tag of the preceding unit and inserts support code into the appropriate AST levels, such as class definitions aligning with the Class Node in the test case. After generating the code structure, LLMCAPLENS processes code semantic by handling comments and renaming variables and functions. Comments from Basic Units, tagged for specific purposes, are selectively retained or removed (e.g., keeping `<CTAG:T>` for correct comment or `<CTAG:F>` otherwise). Variable and function names are replaced with meaningful alternatives (e.g., `untaint_`, `propagateTaint_`), and `EXIT()` is renamed contextually (e.g., `sink()`).

C. Automated Test Case Verification

While prior phases provided test cases, manual labeling remains labor-intensive, particularly for thousands of complex cases. Existing taint analysis tools are unsuitable for automated verification due to their poor performance on complex scenarios (e.g., field sensitivity and loops) or lack of scalability (e.g., taintDroid [29] relies on Dalvik).

To address these challenges, we introduce a lightweight dynamic taint analysis-based approach. The key insight of verification is to infer long-range dataflow facts using precise short-range dataflow facts (i.e., the [RULE] tag), a principle widely adopted in summary-based analyses [20]. We ensure the success of this approach based on the following points: (1) Short-range dataflow facts are at the statement level and remain unaffected during units combinations. (2) Short-range dataflow facts are manually annotated, ensuring accuracy, and the limited number of units and one-time effort minimize manual work. (3) By using dynamic taint analysis, we avoid

```
[case1]: // Nested statements
  VARNAME_1 = source();
  for(int VARNAME_2 = 0; VARNAME_2<1; VARNAME_3 = VARNAME_1){
      sink(VARNAME_3);
      VARNAME_2++;
      [RULE] VARNAME_3.tag = VARNAME_1.tag;}
[case2]: // Functions without source code
  String VARNAME_1 = VARNAME_2.substring(VARNAME_3, VARNAME_4);
  [RULE] if(VARNAME_3 < VARNAME_4)
  [RULE]     VARNAME_1.tag |= VARNAME_2.tag
[case3]: // Java reflection mechanism
  Class<?> VARNAME_2 = Class.forName(VARNAME_1);
  VARNAME_4 = VARNAME_2.getDeclaredMethod
                          (VARNAME_3, String.class);
  VARNAME_8 = VARNAME_4.invoke(VARNAME_7,
                                VARNAME_5, VARNAME_6);
  [RULE] if(VARNAME_1.equals("java.lang.String") &&
                                VARNAME_3.equals("substring")){
  [RULE]      if(VARNAME_5 < VARNAME_6){
  [RULE]          VARNAME_8.tag |= VARNAME_7.tag;}}
  [RULE] if(VARNAME_1.equals("...") && VARNAME_3.equals("...")){
  [RULE]      ... }
[case4]: // Virtual calls
  String VARNAME_2 = VARNAME_1.substring(VARNAME_3, VARNAME_4);
  [RULE] if(VARNAME_1.getClass().getName().
                        equals("java.lang.String")){
  [RULE]    if(VARNAME_3 < VARNAME_4){
  [RULE]        VARNAME_2.tag |= VARNAME_1.tag;
  [RULE] }}
  [RULE] if(VARNAME_1.getClass().getName().equals("org.
                    example.myString")){
  [RULE]    ...}
```

Fig. 5: [RULE] under complex scenarios

the path explosion issues and other accuracy challenges common in static analysis. Specifically, we implemented custom wrapper subclasses for each class (e.g., IntegerWrapper for int), which will be utilized within the [RULE] in the Basic Unit. These subclasses include a tag field to track taint status, propagating tags recursively when set to true. During the automated verification, LLMCAPLENS replaces original classes with these custom wrappers, removes [RULE] strings, and executes the test cases. After execution, LLMCAPLENS inspects sink point parameters and their fields for true tags to detect taint flows. Tag values can also be printed at [RULE] points to trace taint propagation. All modifications are limited to the verification phase, with wrapper classes s and rules removed in the measurements phase.

An important question that arises here is whether we can write accurate rules for sufficiently complex basic units. We believe this is feasible. In Figure 5, we present several complex cases that demonstrate the following advantages of [RULE]: (1) Manually written rules can generate highly concise taint propagation rules for complex programs and functions without source code (case 1, case 2). (2) [RULE] can access the runtime context to obtain precise information (case 3, case 4). (3) [RULE] can utilize code structures such as if branches to achieve more accurate taint propagation (case 3, case 4).

Besides, our approach accounts for several key factors to ensure robustness and practicality. For instance, to reduce frequent rule updates when importing new functions without source code, our Basic Units cover a broader range of commonly used library functions than well-known benchmarks. Additionally, reflection and virtual calls are handled using an enumeration approach. While this method has limitations, it

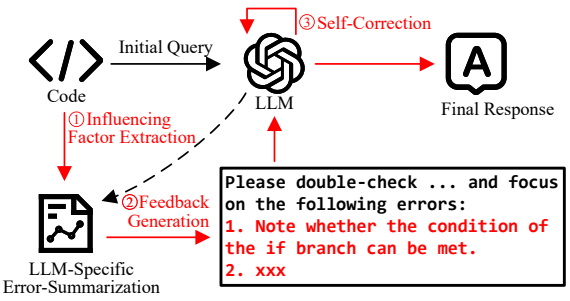

Fig. 6: Error-summarization-based Self-correction Process

aligns with practical scenarios where the number of reflective and virtual calls is typically manageable.

D. Measurement-Driven Enhancement

LLMCAPLENS employs a measurement-driven enhancement approach based on measurement results. LLM self-correction has shown strong performance in recent studies [30], [31]. However, they are not directly applicable for taint analysis tasks, as they rely on case-specific feedback from external validation tools (e.g., compilers), which is impractical in this task. To address this, LLMCAPLENS introduces a measurement-driven method, leverages LLM-specific error causes identified during measurement to prompt LLMs to self-correct mistakes. The key insight is that an LLM tends to repeat similar errors under similar influencing factors.

Figure 6 illustrates the workflow of the measurement-driven self-correction method. First, we manually analyze measurement results, identifying error causes for each influencing factor to derive LLM-specific error summarization. This is a one-time effort, and we sampled 200 responses from the results for analysis, which required approximately seven hours for a security expert with 3 years' experience. Then, when encountering a taint analysis task, LLMCAPLENS extracts influencing factors via traditional static analysis tools(e.g., Joern [32]). Next, LLMCAPLENS uses these factors along with previous Error-Summarization to predict likely LLM errors, and generates corresponding correction prompts such as 'Note if the if branch condition is satisfiable'. These prompts are then concatenated into a self-correction template, guiding the LLM to double-check its answers and produce the final response. This method is training-free and model-specific, and avoids problems in traditional self-correction methods like relying on feedback from external tools [33], [34].

III. MEASUREMENT EVALUATION

We apply LLMCAPLENS to 10 mainstream LLMs, generating benchmarks and revealing LLMs' perform under various influencing factors and identifying unique characteristics,

A. Measurement Setup

In this paper, we propose three measurement plans, including single-factor, multi-factor, and semantic factor plans to comprehensively measure LLMs' taint analysis capabilities. Notably, users can fully design their own plans and use LLMCAPLENS to generate the corresponding benchmarks.

Single-Factor Measurement Plan. This plan aims to measure LLMs' taint analysis capability on individual factors. The design of the measurement plan considers the following aspects:

- Depth of Factors Overlap: Since the number of units directly affects complexity, depth here refers to the number of basic units used. As mentioned in §I, code length is critical due to the LLMs' attention mechanism, measuring how LLMs' performance changes as the depth increases is essential. Considering the complexity of real-world code, the initial depth is 9; if the LLM excels, it increases further.
- Randomness: Due to LLMs' probabilistic nature, their randomness cannot be fully eliminated [35], [7]. LLMCAPLENS addresses it using a voting mechanism, i.e., asking three times and selecting the most frequent response.
- Potential Biases: Previous studies have pointed out that LLMs may exhibit biases, often favoring certain types of responses [7], [8]. To minimize the impact, it is essential to balance the number of positive and negative test cases.

The Algorithm 1 illustrates the details of this plan. During test case generation, operator type (i.e., positions of [EXIT]) is chosen randomly when combining two units.

Multi-Factor Measurement Plan. This plan measures LLMs' performance across various factor combinations, focusing on whether certain combinations significantly affect performance.

In theory, this requires generating and testing all possible factor combinations. While LLMCAPLENS can automate this process, the number of potential combinations—$3 * \sum_{n=1}^{19}\sum_{m=1}^{9} C_{20}^{m} * n^{m}$—reaches $1.32 * 10^{16}$. Given the impracticality of exhaustively testing such a large set within a finite time and budget, this plan focuses on pairwise combinations. However, LLMCAPLENS is flexible and can easily be scaled to include more combinations if resources allow.

Specifically, LLMCAPLENS generates test cases each time using two different influencing factors. For example, by sequentially adding For-Loop$_1$, If-Branch, and For-Loop$_1$ Factors to the case, a Multi-Factor test case with depth 3 is created. Other details mirror the single factor measurement plan, involving mitigating randomness and potential biases.

Semantic Factor Measurement Plan. The semantic factor measurement plan extends the single-factor plan by modifying semantic elements (i.e., variable names, function names, and comments) in generated test cases. Specifically, for comments (e.g., "varname_111 is sanitized"), LLMCAPLENS uses three schemes: correct comments, no comments, and incorrect comments. For variable and function naming, the schemes include taint-indicative names (e.g., "taint_"), meaningless names, and non-taint-indicative names (e.g., "sanitized_"). More details are presented in Table II. These task-specific taint-hinting comments and names are meaningful: on one hand, they follow a common practice for measuring LLMs' capabilities—for example, in prior work [7], variable names were set to patterns such as 'vulnerability-related keywords', and we adapted it to taint analysis; on the other hand, such hints are also prevalent

Algorithm 1: Single-Factor Measurement Plan

Input: LLM Under Evaluation LLM
Input: Influencing Factors $influencingFactors$
Output: Measurement Result $result$

1 $votingNums$ = 3, $depth$ = 9, $result$ = map()
2 **for each** $factor$ *in* $influencingFactors$ **do**
3 $positiveNums$ = *calculatePositiveNums()*
4 $negativeNums$ = *calculateNegativeNums()*
5 $flag_P$, $flag_N$ = *false*, $order$ = 0
6 **while** ***true*** **do**
7 $order$ = $order$ + 1
8 $TPNums$, $TNNums$ = 0
9 **for** $testNum \leftarrow 1$ **to** $positiveNums$ **do**
10 $testCase$ = *posCaseGen(order,factor)*
11 $voteCorrect$ = 0
12 **for** $voteNum \leftarrow 1$ **to** $votingNums$ **do**
13 $response$ = *inquery(LLM,testCase)*
14 **if** *isCorrect(response)* **then**
15 $voteCorrect$ = $voteCorrect$ + 1
16 **end**
17 **end**
18 **if** $voteCorrect > votingNums/2$ **then**
19 $TPNums$ = $TPNums$ + 1
20 *continue*
21 **end**
22 $flag_P$ = *true*
23 **end**
24 **for** $testNum \leftarrow 1$ **to** $negativeNums$ **do**
25 $testCase$ = *negCaseGen(order, factor)*
26 /** Same as lines 11–17 **/
27 **if** $voteCorrect > votingNums/2$ **then**
28 $TNNums$ = $TNNums$ + 1
29 *continue*
30 **end**
31 $flag_N$ = *true*
32 **end**
33 $result[factor][order]$ = (positiveNums, negativeNums, TPNums, TNNums)
34 **if** $order > depth$ *and* $(flag_N$ *or* $flag_P)$ **then**
35 break
36 **end**
37 **end**
38 **end**
39 return *result*

TABLE II: Semantic Factor Measurement Plans

Factor Type	Modification Operation
Comments	Insert correct comments Insert no comments Insert incorrect comments
Naming	Replace with names that suggest the presence or propagation of taint. Replace with meaningless names Replace with names that suggest the absence or sanitization of taint.

in real-world security tasks, and not reliable—for instance, in RuoYi [36] (with 7.5K stars on GitHub), 95.73% of the public interfaces contain the string `RequiresPermission`, yet over-privilege vulnerabilities (e.g., CVE-2025-28407) still exist in those interfaces. Similarly, in Tomcat [37] and Jetty [38], exists many comments such as '// Read data' indicating whether a variable is associated with user input (i.e., taint), and ' // have some safety checks' indicating a sanitizer.

LLM Selection. We selected 10 representative LLMs including GPT-4 Turbo, GPT-3.5 Turbo, Qwen2.5-72B Instruct, Spark-Max, DeepSeek-R1 671B, DouBao-Pro 32k, Calme-2.1-Qwen2-72B, Code Llama 70B, Llama3 70B, and Claude 3 Opus, as detailed in Table IV. Complete results are provided along with the source code.

Parameter Settings. Taint analysis requires rigorous logical reasoning; thus, *Temperature* is set to 0, and *Top-p* retains its default value, as in prior studies [35], [39], [7].

Prompt Template. To objectively measure the taint analysis capability of LLMs, we use a prompt template that combines advanced techniques such as few-shot, task-oriented, chain of thought (CoT), and 'mimic-in-the-background' [2], as shown in Figure 7. These methods, shown effective in prior work [2], reduce randomness and improve performance.

System Prompt

```
I will give you some code snippets to help me
determine if there is any data flow from the
source() function to the parameters of the sink()
function.
You can mimic answering them in the background five
times and provide me with the most frequently
appearing answer. Be careful not to assume anything
beyond the code.
```

User Prompt

```
Analysis:
   code:
   "
   String source = source();
   String taint = source;
   sink(taint);
   "
   DataFlow From source() To sink(): [True]
Analysis:
    code:
    $code under analysis$
1. You simulate the program execution to analyze
its data flow and tell me the result in the form
[True] or [False] on a separate line at the end.
```

Fig. 7: General Prompt Template Used in Evaluation.

Effectiveness of Automated Verification. We used a lightweight dynamic taint analysis technique to implement automatic verification while ensuring the diversity of our benchmark. We implemented dynamic taint analysis with 1.5K LOC Python code. We randomly selected 200 test cases from the dynamic taint analysis results, all of which were correct.

B. Benchmark Distribution

Following the plans above, we generated a total of 30,704 unique test cases, each test case contains 8–1036 LoC, uniformly distributed. The LoC of each test case is equal to the sum of the code snippet LoC in the Basic Units used to generate it. We also tested our benchmark with the SOTA tool Tai-e [17], achieving 72.24% accuracy but a 54.93% false positive rate. Table III shows the distribution of Single-Factor benchmark and Tai-e's result. Overall, Tai-e shows a higher TPR than TNR, consistent with many static analyses that emphasize soundness. These analyses often use over-approximation to reduce false negatives, increasing false positives. Meanwhile, Tai-e performs poorly on factors such as `Constant-Folding` and `Index-Sensitive`, which require fine-grained reasoning about value ranges, index bounds, and cross-path constraints, because to mitigate issues such as path explosion and maintain scalability, traditional tools often

rely on coarse over-approximation (e.g., marking the entire array as tainted if one element is), sacrificing precision. In contrast, LLMs can leverage contextual semantic understanding to perform implicit pruning and related effects, improving accuracy and complementing the limitations of traditional tools [3], [4]. For deterministic problems like object sensitivity, Tai-e outperforms LLMs. Besides, compared to OWASP(2,700+ test cases), our benchmark includes four additional factors(Const-Folding, Static Propagation, Object-Sensitive, Alias-Sensitive) and 27,964 test cases.

TABLE III: The Distribution of Single-Factor Benchmark

Influencing Factor	Numbers	LoC	**Tai-e TPR**	**Tai-e TNR**
For-Loop$_1$	190	10-219	100.0%	0.0%
For-Loop$_2$	190	12-210	100.0%	77.78%
If-Branch	190	10-211	100.0%	62.96%
Sequential-Sensitive	76	8-134	100.0%	100.0%
Switch-Branch	114	15-396	100.0%	72.22%
WhileLoop$_1$	190	12-262	100.0%	0.0%
WhileLoop$_2$	190	14-248	100.0%	75.0%
If-Path-Sensitive	114	13-340	100.0%	88.89%
Switch-Path-Sensitive	114	15-312	100.0%	75.93%
Constant-Folding	114	9-208	100.0%	0.0%
String-Branch	76	14-248	100.0%	0.0%
Context-Sensitive	76	10-226	100.0%	92.59%
Field-Sensitive	76	12-264	100.0%	44.28%
Static Propagation	76	14-338	100.0%	0.0%
List-Index-Sensitive	114	13-229	100.0%	0.0%
Alias-Sensitive	76	20-417	100.0%	0.0%
Invoke-Sensitive	76	53-1036	88.89%	66.67%
Map-Index-Sensitive	76	12-210	100.0%	0.0%
Obj-Sensitive	114	12-264	100.0%	100.0%

C. Single-Factor Measurement

As outlined in §III-A, LLMCAPLENS generates 2,242 test cases for single-factor measurement and reports LLMs' accuracy, TPR (True Positive Rate), and TNR (True Negative Rate). The overview of all LLMs' results is shown in Table IV, and Table V presents the detailed results for GPT-4 Turbo.

Finding 1: LLMs struggle to balance accuracy and time. Some LLMs like deepseek-r1, show better performance but take 22 seconds per test case average and the longer exceeding 300 seconds, making them impractical for real-world projects. Conversely, faster models like GPT-4 Turbo sacrifice accuracy, with a TNR of only 49.66%. This demonstrates that directly applying LLMs to real-world code is not feasible.

Finding 2: LLMs exhibit various biases. Some LLMs show accuracy of roughly 50% across factors, indicating they repeat the same answer, suggesting strong output bias. Most large models, like GPT-4 turbo, prioritize "soundness" in taint analysis and tend to provide positive answers, similar to traditional tools [32]. In contrast, Doubao focuses on minimizing false positives and accepting some false negatives.

Finding 3: LLMs share similar strengths and weaknesses. Measurement shows LLMs excel at handling factors like 'If-Branch' and 'Sequential-Sensitive' but struggle with factors like 'For-Loop$_1$' and 'While-Loop$_1$'. This may be due to the need for stronger reasoning capabilities when dealing with loops, such as account for cumulative effects, as LLMs cannot simulate each iteration.

D. Multi-Factor Measurement

LLMCAPLENS follows the Multi-Factor Measurement Plan outlined in §III-A to generate 19,494 test cases involving 171 combinations for measurementing LLMs.

Finding 4: LLMs largely align across single and multi-factor scenarios. Multi-factor measurements largely mirror single-factor results. For example, LLMs still exhibit various biases and are unsuitable for direct application. This suggests that these characteristics are related to the inherent mechanisms of LLMs.

Finding 5: Few combinations increase problem difficulty. Most combinations have minimal impact on LLM performance. For instance, Qwen2.5-72B Instruct performs between its results on Switch-Path-Sensitive and Map-Index-Sensitive. However, combinations like Context-Sensitive and For-Loop$_1$ significantly affect performance, likely due to the added complexity of loop semantics.

E. Semantic Factor Measurement

Following the plan in §III-A, LLMCAPLENS integrates semantic information into test cases to assess its impact. Results for Qwen2.5-72B Instruct are shown in Table VI.

Finding 6: Function and variable names have minimal impact. When function and variable names are changed to names with taint-indicative semantics, Qwen2.5-72B Instruct shows a 3.22% increase in TPR and a 7.05% decrease in TNR. Similarly, changing their names to non-taint-indicative semantics results in a 2.63% decrease in TPR and a 4.84% decrease in TNR. This is consistent with previous works [40].

Finding 7: Code comments have a significant impact. After adding correct code comments, such as "function_4444 will not be executed" and "varname_3 is sanitized", the TNR of Qwen2.5-72B Instruct increased by an average of 45.16%. On the other hand, after introducing incorrect comments, the Qwen2.5-72B Instruct's TPR decreased by an average of 76.26%. Note that incorrect comments can also lead to an increase in TNR, this may be because incorrect comments reverse LLM's bias, leading to more negative answers.

IV. ENHANCEMENT EVALUATION

This section shows how LLMCAPLENS enhances LLM's performance in taint-analysis and downstream applications, i.e., vulnerability detection. We enhance LLMs based on single-factor measurement results, as they are more atomic, and experiments demonstrate it is sufficiently effective.

A. Summary of LLM taint analysis errors.

By analyzing the reasoning process of LLM on incorrect test cases, we categorize their errors into four general types: Lack of focus, Reasoning bias, Hallucinations or lack of knowledge, and "Well-intentioned" assumption. Table VII presents the distribution of error types for GPT-4 Turbo as an example.

- *Lack of focus.* In some cases, LLMs may overlook key details, such as whether a branch condition is satisfied.
- *Reasoning bias.* LLMs may correctly understand program details but err in reasoning, such as recognizing an unmet if condition yet incorrectly concluding the if block executes.

TABLE IV: The Overview of All LLMs Performance on Single Factor Measurement

Factor	Gpt-4 Turbo	Gpt-3.5 Turbo	Qwen2.5-72B Instruct	Spark-Ultra	DeepSeek-R1 671B *	DouBao Pro 32K	Calme-2.1 Qwen2-72B	Llama3 70b	Code Llama 70B †	Claude 3 Opus
					Accuracy					
ForLoop_1	47.50%	53.00%	53.78%	50.00%	97.36%	50.00%	55.56%	47.22%	19.45%	47.00%
ForLoop_2	68.00%	73.50%	61.11%	50.00%	85.97%	50.00%	76.39%	50.00%	11.11%	51.50%
If-Branch	83.50%	79.50%	62.97%	50.00%	98.33%	50.00%	62.97%	53.71%	20.37%	55.50%
Sequential-Sensitive	100.00%	78.00%	83.34%	55.55%	97.50%	50.00%	94.45%	83.34%	16.67%	78.00%
Switch-Branch	61.00%	58.00%	50.00%	50.00%	73.19%	50.00%	52.78%	52.78%	5.56%	55.50%
While-Loop_1	50.00%	53.00%	52.77%	50.00%	69.86%	50.00%	52.78%	50.00%	11.11%	55.50%
While-Loop_2	69.50%	58.50%	63.89%	51.39%	94.93%	50.00%	70.83%	52.78%	11.11%	58.50%
If-Path-Sensitive	81.50%	61.00%	79.63%	53.71%	78.14%	50.00%	62.97%	59.26%	5.56%	89.00%
Switch-Path-Sensitive	78.00%	58.50%	50.00%	50.00%	83.47%	50.00%	50.00%	52.78%	11.11%	80.50%
Constant-Folding	78.00%	78.00%	68.52%	50.00%	100.00%	50.00%	70.37%	51.85%	5.56%	81.50%
String-Branch	78.00%	78.00%	50.00%	50.00%	97.50%	55.55%	50.00%	50.00%	5.56%	55.50%
Context-Sensitive	72.00%	61.00%	61.11%	50.00%	85.00%	55.55%	55.55%	66.66%	11.11%	61.00%
Field-Sensitive	66.50%	61.00%	55.55%	50.00%	95.00%	55.55%	55.55%	50.00%	11.11%	61.00%
Static Propagation	94.50%	89.00%	61.11%	50.00%	97.50%	50.00%	77.78%	61.11%	16.67%	100.00%
List-Index-Sensitive	75.00%	61.00%	52.77%	50.00%	92.50%	50.00%	58.33%	58.34%	11.11%	80.50%
Map-Index-Sensitive	78.00%	72.00%	83.34%	50.00%	85.00%	50.00%	94.45%	94.45%	11.11%	94.50%
Object-Sensitive	61.00%	61.00%	55.55%	50.00%	92.50%	50.00%	55.55%	50.00%	0.00%	72.00%
Alias-Sensitive	100.00%	94.50%	88.89%	50.00%	100.00%	61.11%	88.89%	83.34%	11.11%	100.00%
Invoke-Sensitive	58.00%	55.50%	27.78%	22.22%	58.61%	52.78%	44.45%	52.78%	0.00%	75.00%
Average	72.00%	67.58%	60.83 %	49.14%	88.54%	51.53%	63.56%	58.52%	10.60%	71.16%

† CodeLlama 70B refused to answer 447/594 test cases due to ethical concerns that do not actually exist, resulting in a significantly low accuracy.
* DeepSeek-R1 has a stronger taint analysis capability, so its test depth ranges from 1 to 20 rather than 9.

- *Hallucinations or lack of relevant knowledge.* Despite their strong generalization and knowledge, LLMs can still hallucinate or lack key knowledge in complex cases, such as wrongly assuming the array's index starts at 1 instead of 0.
- *"Well-intentioned" assumptions.* LLMs naturally correct errors like typos, but this can sometimes be a drawback. For instance, in field-sensitivity cases, an LLM may distinguish fields correctly but mistakenly treat them as typos.

B. Enhancement on Individual Factors

We designed factor-specific self-correction prompts based on the underlying causes for each LLM, following subsection II-D. This is a one-time effort with reusable results.

Self-Correction Prompts. Part of these prompts for GPT-4 Turbo is shown in Table VIII. A prefix will be added such as "Double-check the results ... and focus on following errors:".

Enhancement Results. We observed a significant improvement in their taint analysis capability. Table IX presents the enhancement results for GPT-4 Turbo on various taint analysis factors. Our method improves accuracy on 16 of 19 factors, boosting average TNR from 49.66% to 70.95%, with a notable increase for WhileLoop_1 from 16.67% to 61.11%. Despite a slight TPR decrease, the significant TNR gain makes it acceptable. This results from the self-correction process providing potential errors, which may occasionally be misclassifying.

C. Enhancement on Vulnerability Detection

Dataset. We randomly selected 30 candidates, including 15 vulnerable samples and their patched versions, from the well-known benchmark NIST-SARD, based on real-world[9], [41].

Experiment Design. We compare four methods for vulnerability detection:

- Method-1 directly queries the LLM about the presence of a specific vulnerability. We use the most effective prompts proposed in previous work[7].
- Method-2 instructs the LLM to analyze the program, first checking for a feasible data flow between the source and sink related to the vulnerability, then determining its presence as in Method-1.
- Method-3 extends Method-2 by incorporating potential LLM-specific errors into the prompt, providing information about possible mistakes alongside the code for analysis.
- Method-4 uses a self-correction approach with LLM-specific errors (as detailed in §II-D). Rather than embedding errors in the prompt, it provides them as feedback for self-correction.

Experiment Results. Table X shows the results for the four methods, with Method-4 yielding the best results. Compared to Method-1, Method-4 achieves a 33.34% accuracy improvement, with a notable 73.3% improvement in patched code. This highlights that our approach enhances the LLM's sensitivity to subtle code changes, such as array index modifications or added if statements. Method-2 results show that merely adding taint analysis does not significantly improve the performance, while the comparison between Method-4 and Method-3 demonstrates the effectiveness of the self-correction.

V. Discussion and Limitation

Dataset Generation. LLMCapLens automates test case generation and verification, creating a dedicated taint analysis dataset. This dataset can further be used to evaluate the effectiveness of traditional tools. Moreover, LLMCapLens allows fine-tuning of various LLMs by generating datasets tailored to their capabilities. While primarily focusing on binary results—aligning with security developers' output priority—LLMCapLens can also generate benchmarks with reasoning steps by tracking taint traces at [RULE] points. LLMCapLens can also be extended to create datasets for other tasks, e.g., symbolic execution.

TABLE V: Single-factor measurement results for GPT-4 Turbo. The boundary between the blue and orange blocks represents accuracy, with the red squares indicating TNR and the green squares indicating TPR.

TABLE VI: Overview of Semantic Factor Measurement for Qwen2-72B Instruct

Semantic Type	Modify Method	TPR	TNR
Comments	correct comments	93.89%	75.84%
	no comments	91.52%	30.68%
	incorrect comments	15.26%	70.31%
Naming	taint-indicative	94.74%	23.63%
	meaningless	91.52%	30.68%
	non-taint-indicative	88.89%	25.84%

TABLE VII: Distribution of Error Types in GPT-4 Turbo

Error Type	Taint Analysis Influencing Factor
Lack of focus	ForLoop$_1$, ForLoop$_2$, If-Branch, Switch-Branch, Sequential-Sensitive, While-Loop$_1$, While-Loop$_2$, Switch-Path-Sensitive, String-Branch, Context-Sensitive, Field-Sensitive, Static Propagation, Object-Sensitive
Reasoning bias	ForLoop$_2$, Switch-Branch, While-Loop1, While-Loop1, If-Path-Sensitive, Switch-Path-Sensitive, List-Index-Sensitive
Hallucinations or lack knowledge	List-Index-Sensitive
"Well-intentioned" assumption	Map-Index-Sensitive, Field-Sensitive, Object-Sensitive, Alias-Sensitive, Invoke-Sensitive

Completeness of influencing factors and basic generation units. Although we have extensively studied and summarized the influencing factors and Basic Units, guaranteeing the theoretical completeness is challenging. However, LLMCapLens is highly extensible, readily accommodating future additions of influencing factors and units.

Manual Effort and Automation. LLMCapLens requires manual effort primarily for constructing Basic Units and summarizing self-correction prompts, both of which are one-time tasks with low overhead, e.g., 118 Basic Units yield over 10^{16} test cases. Furthermore, the prompt summarization can be automated by introducing an LLM to identify errors in original output and generate appropriate self-correction prompts.

VI. Related Work

LLM Capability Understanding and Measurement. LLMs have demonstrated remarkable potential across various domains, prompting growing interest in investigating their capabilities in specialized areas. For example, in code-related tasks, researchers have explored LLMs' effectiveness in vulnerability detection [7], [8], code understanding [40], [35], [42], code generation [43], [44], code repair [45], [46], and program state inference [47]. While these studies have yielded some intriguing insights, they primarily aim to qualitatively determine whether LLMs can be directly applied, without delving into the underlying factors that influence their performance. In this paper, we conduct the first systematic study of LLMs' taint analysis capability, providing a quantitative evaluation across various dimensions through comprehensive modeling and automated test case generation and verification.

TABLE VIII: Part of Self-Correction Prompts for GPT-4 Turbo.

Factor	Correction Prompt
For-Loop$_1$	1. Misjudging the value of the variable during the last iteration of the loop. 2. Don't make any assumptions, just follow the code.
If-Branch	1. Note whether the condition of the if branch can be met.
Switch-Branch	1. Note to choose the correct switch path. 2. Note that the statement after `break` should not be executed.
WhileLoop$_1$	1. Note that each iteration of the loop changes the value of one of the variables in the loop. 2. Notice the value of the variable in the last loop.
WhileLoop$_2$	1. Notice the value of the variable during the last iteration of the loop 2. Notice if sink() gets executed, and only when the condition is met the dataflow result is true.
If-Path-Sensitive	1. Note that if and else cannot be executed at the same time.
String-Branch	1. Note whether the condition of the if branch can be satisfied.
Field-Sensitive	1. Note the difference between fields within the same class.
List-Index-Sensitive	1. Note how the index changes after adding or removing elements. 2. Note the index starts at 0.

TABLE IX: Single-Factor Enhanced Result on GPT-4 Turbo.

Factor	GPT-4		GPT-4 Enhanced	
	TPR	TNR	TPR	TNR
For-Loop$_1$	88.89%	5.56%	83.33%	50.00%↑
For-Loop$_2$	94.44%	41.67%	88.89%	63.89%↑
If-Branch	100.00%	66.67%	88.89%	70.37%↑
Sequential-Sensitive	100.00%	100.00%	100.00%	100.00%
Switch-Branch	100.00%	22.22%	100.00%	44.44%↑
WhileLoop$_1$	83.33%	16.67%	72.22%	61.11%↑
WhileLoop$_2$	100.00%	38.89%	83.33%	58.33% ↑
If-Path-Sensitive	100.00%	62.96%	89.89%	74.07%↑
Switch-Path-Sensitive	100.00%	55.56%	100.00%	72.22%↑
Constant-Folding	100.00%	55.56%	100.00%	92.59%↑
String-Branch	100.00%	55.56%	100.00%	77.78%↑
Context-Sensitive	100.00%	44.44%	100.00%	66.67%↑
Field-Sensitive	100.00%	33.33%	100.00%	55.56%↑
Static Propagation	100.00%	88.89%	100.00%	88.89%↑
List-Index-Sensitive	94.44%	55.56%	88.89%	72.22%↑
Map-Index-Sensitive	100.00%	55.56%	78.78%	88.89%↑
Obj-Sensitive	100.00%	22.22%	100.00%	66.67%↑
Alias-Sensitive	100.00%	100.00%	100.00%	100.00%
Invoke-Sensitive	94.44%	22.22%	94.44%	44.44%↑
Average	97.66%	49.66%	92.98%	70.95%

LLM for Security Analysis. Several studies have explored integrating LLMs into traditional security solutions to enhance their capability. For instance, in the field of static analysis [5], [2], [3], [4], Cao et al.[5] utilized LLM for fine-grained analysis of vulnerabilities in cloud applications, Sun et al.[2] utilized LLMs to assist in detecting business logic vulnerabilities in smart contract code, while Li et al.[3], [4] employed LLMs to analyze the initialization states of variables, aiding in the detection of Use-Before-Initialization vulnerabilities. In the field of fuzzing[48], [49], [50], [51], [52], Deng et al.[48] leveraged LLMs' understanding of library code to improve fuzz driver generation, and Meng et al.[50], [53] used LLMs' comprehension of documentation to generate high-quality inputs for IoT protocol fuzzing. In contrast, our work aims to systematically analyzes and enhances LLMs' intrinsic taint analysis capabilities. Our method advances LLM-based security solutions by improving their taint analysis capability, thereby enabling more effective and broader applications.

TABLE X: Vulnerability detection performance of four methods. The 'v' represents vulnerable code, while 'p' represents the patched code.

Vulnerability	LoC†	Method-1		Method-2		Method-3		Method-4	
		v	p	v	p	v	p	v	p
v156009df07cf07	634	✓	✗	✓	✗	✓	✗	✓	✓
v156320df18cf18	432	✓	✗	✓	✗	✓	✗	✓	✓
v156393df12cf12	384	✓	✗	✓	✗	✓	✗	✓	✓
v156511df06cf06	552	✓	✗	✓	✓	✓	✗	✓	✓
v155391df07cf07	524	✓	✗	✗	✓	✓	✗	✓	✓
v155409df18cf18	570	✓	✗	✓	✗	✓	✗	✓	✓
v155462df06cf06	203	✓	✓	✓	✓	✓	✓	✓	✓
v156314df11cf01	210	✓	✗	✗	✓	✓	✗	✓	✓
v156349df07cf07	463	✓	✗	✓	✗	✓	✗	✓	✗
v156377df06cf06	309	✓	✗	✓	✗	✓	✗	✓	✓
v156396df18cf18	298	✓	✗	✓	✗	✓	✓	✓	✓
v155366df11cf11	656	✓	✗	✓	✗	✓	✗	✓	✗
v156230df11cf11	713	✓	✗	✓	✗	✓	✓	✓	✓
v156074df11cf11	722	✓	✗	✓	✗	✓	✗	✓	✗
v156002df05cf05	438	✓	✗	✓	✗	✓	✗	✗	✓
Accuracy	473.87	53.33%		56.67%		60%		86.67%	

† LoC of the vulnerable code, similar to the patched version.

VII. Conclusion

In this paper, we present a dynamic benchmark generation framework, LLMCapLens, to quantitatively and comprehensively measure the taint analysis capability of large language models (LLMs) and enable targeted improvements. By applying LLMCapLens to 10 mainstream LLMs, we gain insights into their taint analysis performance across various code structural and semantic factors. Based on the analysis of LLM-specific errors, LLMCapLens enhances LLMs using a self-correction method derived from error summarization, improving their capability. Experiments demonstrate our approach substantially enhances the effectiveness of LLMs in taint analysis-based tasks like vulnerability detection. For further research, we have released the code and dataset at https://github.com/HRsGIT/LLMCapLen.

Acknowledgement

We would like to thank anonymous shepherd and reviewers for their helpful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (62102093, U2436207, 62172104, 62172105, 62202106, 62302101, 62102091, 62472096, 62402114, 62402116). Min Yang is the corresponding author and a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012).

REFERENCES

[1] G. Lu, X. Ju, X. Chen, W. Pei, and Z. Cai, "Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning," *Journal of Systems and Software*, vol. 212, p. 112031, 2024.

[2] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[3] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Assisting static analysis with large language models: A chatgpt experiment," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 2107–2111.

[4] ——, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, 2024.

[5] D. Cao and W. Jun, "Llm-cloudsec: Large language model empowered automatic and deep vulnerability analysis for intelligent clouds," in *IEEE INFOCOM 2024-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2024, pp. 1–6.

[6] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[7] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, "Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks," in *IEEE Symposium on Security and Privacy*, 2024.

[8] N. Risse and M. Böhme, "Uncovering the limits of machine learning for automatic vulnerability detection," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4247–4264.

[9] NIST, "Nist software assurance reference dataset," https://samate.nist.gov/SARD/, July, 2024.

[10] OWASP, "Owasp benchmark," https://owasp.org/www-project-benchmark/, July, 2024.

[11] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, 2009, pp. 243–262.

[12] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "A principled approach to selective context sensitivity for pointer analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 42, no. 2, pp. 1–40, 2020.

[13] W. Ma, S. Yang, T. Tan, X. Ma, C. Xu, and Y. Li, "Context sensitivity without contexts: A cut-shortcut approach to fast and precise pointer analysis," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 539–564, 2023.

[14] T. Tan, Y. Li, and J. Xue, "Efficient and precise points-to analysis: modeling the heap by merging equivalent automata," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 278–291.

[15] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Precision-guided context sensitivity for pointer analysis," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[16] Y. Zhai, Y. Hao, Z. Zhang, W. Chen, G. Li, Z. Qian, C. Song, M. Sridharan, S. V. Krishnamurthy, T. Jaeger *et al.*, "Progressive scrutiny: Incremental detection of ubi bugs in the linux kernel," in *2022 Network and Distributed System Security Symposium*, 2022.

[17] T. Tan and Y. Li, "Tai-e: A developer-friendly static analysis framework for java by harnessing the good designs of classics," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1093–1105.

[18] 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*, 2020, pp. 221–232.

[19] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Highly precise taint analysis for android applications," 2013.

[20] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995, pp. 49–61.

[21] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang, "Smoke: scalable path-sensitive memory leak detection for millions of lines of code," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 72–82.

[22] M. Das, S. Lerner, and M. Seigle, "Esp: Path-sensitive program verification in polynomial time," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002, pp. 57–68.

[23] J. Späth, L. Nguyen Quang Do, K. Ali, and E. Bodden, "Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2016.

[24] J. Lerch, J. Späth, E. Bodden, and M. Mezini, "Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 619–629.

[25] E. Bodden, "The secret sauce in efficient and precise static analysis: The beauty of distributive, summary-based static analyses (and how to master them)," in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018, pp. 85–93.

[26] 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*, 2021, pp. 811–824.

[27] J. Zhao, Y. Li, Y. Zou, Z. Liang, Y. Xiao, Y. Li, B. Peng, N. Zhong, X. Wang, W. Wang *et al.*, "Leveraging semantic relations in code and data to enhance taint analysis of embedded systems," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 7067–7084.

[28] xAST, "xast repository," https://github.com/alipay/ant-application-security-testing-benchmark, October, 2025.

[29] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 1–29, 2014.

[30] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang *et al.*, "Self-refine: Iterative refinement with self-feedback," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[31] E. First, M. N. Rabe, T. Ringer, and Y. Brun, "Baldur: Whole-proof generation and repair with large language models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1229–1241. [Online]. Available: https://doi.org/10.1145/3611643.3616243

[32] Joern, "Joern," https://joern.io/, July, 2024.

[33] R. Kamoi, Y. Zhang, N. Zhang, J. Han, and R. Zhang, "When can llms actually correct their own mistakes? a critical survey of self-correction of llms," *arXiv preprint arXiv:2406.01297*, 2024.

[34] Z. Gou, Z. Shao, Y. Gong, Y. Shen, Y. Yang, N. Duan, and W. Chen, "Critic: Large language models can self-correct with tool-interactive critiquing," *arXiv preprint arXiv:2305.11738*, 2023.

[35] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng, "Exploring the potential of chatgpt in automated code refinement: An empirical study," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[36] RuoYi, "Ruoyi repository," https://github.com/yangzongzhuan/RuoYi, October, 2025.

[37] A. Tomcat, "Apache tomcat repository," https://github.com/apache/tomcat, October, 2025.

[38] E. Jetty, "Eclipse jetty canonical repository," https://github.com/jetty/jetty.project, October, 2025.

[39] P. Liu, J. Liu, L. Fu, K. Lu, Y. Xia, X. Zhang, W. Chen, H. Weng, S. Ji, and W. Wang, "Exploring {ChatGPT's} capabilities on vulnerability management," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 811–828.

[40] C. Fang, N. Miao, S. Srivastav, J. Liu, R. Zhang, R. Fang, R. Tsang, N. Nazari, H. Wang, H. Homayoun *et al.*, "Large language models for code analysis: Do {LLMs} really do their job?" in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 829–846.

[41] NIST, "Nist software assurance reference dataset document," https://samate.nist.gov/SARD/documentation, July, 2024.
[42] 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*, 2024, pp. 1–13.
[43] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at c: A user study on the security implications of large language model code assistants," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2205–2222.
[44] Q. Gu, "Llm-based code generation method for golang compiler testing," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 2201–2203.
[45] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2339–2356.
[46] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1646–1656.
[47] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, "Can large language models reason about program invariants?" in *International Conference on Machine Learning*. PMLR, 2023, pp. 27 496–27 520.
[48] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.
[49] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.
[50] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.
[51] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, 2023, pp. 423–435.
[52] A. Happe and J. Cito, "Getting pwn'd by ai: Penetration testing with large language models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 2082–2086.
[53] Y. Lyu, Y. Xie, P. Chen, and H. Chen, "Prompt fuzzing for fuzz driver generation," in *ACM Conference on Computer and Communications Security (CCS)*, Salt Lake City, UT, USA, 10 2024.