TaintP2X: Detecting Taint-Style Prompt-to-Anything Injection Vulnerabilities in LLM-Integrated Applications

Junjie He*†
hjj@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

Shenao Wang*†
shenaowang@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

Yanjie Zhao†
yanjie_zhao@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

Xinyi Hou†
xinyihou@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

Zhao Liu
r3pwnx@gmail.com
360 AI Security Lab
Beijing, China

Quanchen Zou
zouquanchen@gmail.com
360 AI Security Lab
Beijing, China

Haoyu Wang†‡
haoyuwang@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

ABSTRACT

Large Language Models (LLMs) have revolutionized numerous domains, enabling the development of LLM-integrated applications that autonomously plan and act through tool calling. While these applications demonstrate remarkable capabilities, their ability to invoke sensitive operations, such as file system interactions, code execution, and database queries, introduces critical security risks. In particular, prompt injection vulnerabilities, combined with security-sensitive sink functions, can lead to a broad class of attacks we define as *Prompt-to-Anything Injection (P2Xi)*. These vulnerabilities, stemming from the misuse of LLM-generated outputs without proper validation, can result in severe consequences such as Remote Command Execution (RCE), file injection, SQL injection, and Server-Side Request Forgery (SSRF). To address this emerging threat, we propose TaintP2X, a novel static taint analysis framework that models LLM-generated outputs as taint sources, tracks their propagation through sensitive sink functions, and employs LLM-assisted analysis to prune false positives. TaintP2X achieves high precision and scalability, systematically identifying P2Xi vulnerabilities. In evaluations, TaintP2X demonstrated a 77.1% recall on a ground truth dataset of 35 P2Xi vulnerabilities, outperforming state-of-the-art methods. With TaintP2X, we have uncovered 101 taint paths across 75 open source repositories, with 7 vulnerabilities confirmed by developers, and 5 of them fixed. These findings highlight the prevalence and impact of P2Xi vulnerabilities and establish TaintP2X as a practical solution for securing LLM-integrated ecosystems.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software defect analysis**.

ACM Reference Format:
Junjie He, Shenao Wang, Yanjie Zhao, Xinyi Hou, Zhao Liu, Quanchen Zou, and Haoyu Wang. 2026. TaintP2X: Detecting Taint-Style Prompt-to-Anything Injection Vulnerabilities in LLM-Integrated Applications. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3773199

1 INTRODUCTION

Large Language Models (LLMs) have revolutionized numerous tasks, demonstrating remarkable capabilities in areas such as code generation [11], natural language understanding [25], and logical reasoning [12]. Building on these advancements, **LLM-integrated applications** [42, 48], also referred to as LLM-based agents [41], have emerged as the early forms of neuro-symbolic systems [18], combining LLMs' natural language understanding with structured logic to autonomously plan and act in dynamic environments. One of the most important features of these agents is their ability to perform actions, primarily implemented through tool calling [17, 33, 34] (also called function calling [7, 36]), which enables interactions with local systems and external environments. For instance, frameworks like LangChain [17] and CrewAI [5] have introduced high-level abstractions and APIs to support tool integrations. LangChain, for example, provides a tool decorator that allows

*Both authors contributed equally to this research.
†Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology.
‡Haoyu Wang (haoyuwang@hust.edu.cn) is the corresponding author.

ICSE '26, April 12–18, 2026, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773199

developers to define external functions accessible to the LLM during runtime [17], while CrewAI offers a flexible framework where agents can leverage a wide range of pre-built or custom tools in response to user prompts [5]. Building on these capabilities, LLM-integrated applications have rapidly evolved into a streamlined development paradigm, widely adopted across various domains, including mobile services [40, 43], DeFi [15, 39], and various software engineering tasks [14, 20, 45].

Despite the advancements brought by LLM-integrated applications, the ability to invoke tools that perform sensitive operations, such as file system interactions, code execution, and database queries, also introduces critical security challenges [8, 21, 27], especially in the context of prompt injection vulnerabilities. For instance, Liu *et al.* [21] introduced the concept of *LLM4Shell*, where prompt manipulations enable attackers to execute arbitrary shell commands, while Pedro *et al.* [27] demonstrated *Prompt-to-SQL Injection*, where malicious prompts can lead to unsafe SQL queries in Text-to-SQL systems [31]. However, the risks posed by combining prompt injection with security-sensitive operations extend far beyond these specific examples. To address this broader class of vulnerabilities, we introduce and define a new vulnerability paradigm called **Prompt-to-Anything Injection**, which we refer to as *P2Xi*[1] for short in this paper. These vulnerabilities arise from developers' over-trust in LLM-generated outputs and their failure to validate or sanitize unpredictable contents before passing them to downstream sensitive sink functions. Notably, a specific Common Weakness Enumeration (CWE) category, *CWE-1426: Improper Validation of Generative AI Output*, has been introduced to describe this vulnerability pattern. Depending on the combination of sensitive sink functions involved, these vulnerabilities can result in a wide range of security consequences, such as command injection (leading to Remote Command Execution, RCE), code injection (leading to Arbitrary Code Execution, ACE), SQL injection (SQLi), file injection [4] (leading to arbitrary file read/write, as LLM-generated outputs can manipulate file paths or contents), Server-Side Request Forgery (SSRF, as LLM outputs may construct unsafe URLs to internal services), Cross-Site Scripting (XSS, when unescaped LLM-generated HTML or JavaScript is rendered in the browser), Server-Side Template Injection (SSTI, as LLM-generated template is processed by unsafe template engines), and so on.

Research Gaps. Existing works have made some efforts in identifying these vulnerabilities in LLM-integrated applications, but face notable limitations in scope and scalability. For example, *Pedro et al.* [27] focuses on Prompt-to-SQL injection vulnerabilities, relying on manual testing and evaluating only 5 real-world applications. LLMSmith [21] combines lightweight static analysis and white/black-box testing to detect LLM4Shell vulnerabilities, but its evaluation is limited to 11 frameworks and 50 applications. Building on these efforts, AgentFuzz [8] introduces a directed greybox fuzzing approach to detect taint-style vulnerabilities in LLM-based agents, using LLMs to generate functionality-specific seed prompts and employing feedback-driven prioritization and mutation strategies. However, it remains constrained to dynamic testing, analyzing only 20 agent applications. In summary, current methods suffer from two key limitations. First, they narrowly focus on specific vulnerability types, without addressing the broader P2Xi paradigm. Second, their reliance on dynamic testing limits scalability, as it requires runtime execution of applications and restricts applicability to frameworks and applications with pre-built testing environments. These limitations leave the potential prevalence and impact of P2Xi vulnerabilities in the wild largely unexplored, particularly across the diverse and less-studied applications within the long-tail of LLM-integrated agent ecosystems.

Insights and Challenges. To this end, our core insight is that the sensitive sink functions associated with P2Xi vulnerabilities align with those in traditional taint-style vulnerabilities, but the taint source in P2Xi originates from LLM-generated outputs rather than direct user inputs. This unique characteristic necessitates redefining taint source specifications to model the outputs of LLMs and accurately track their propagation through LLM-integrated applications. Building on this insight, our primary research objective is to extend existing static taint analysis techniques to detect P2Xi vulnerabilities. However, this introduces several key challenges. First, precisely modeling LLM-generated outputs as taint sources is inherently complex, as these outputs are highly context-dependent, influenced by prompts, application logic, and runtime interactions. Second, not all LLM-generated outputs are user-controlled, requiring semantic analysis to identify which outputs are influenced by user prompts and thus represent potential taint sources. Third, detecting P2Xi requires a comprehensive understanding of sanitizer functions along the taint propagation path, demanding semantic analysis to determine whether outputs are adequately validated or sanitized before reaching sensitive sinks.

Our Work. To detect P2Xi vulnerabilities, we propose a novel static taint analysis approach named TaintP2X. First, to handle the complexity of modeling LLM-generated outputs as taint sources, we pre-define a set of standard API interfaces from 49 major LLM providers and identify 48 commonly used LLM gateway interfaces from third-party libraries. Additionally, we leverage large language models to analyze developer-defined wrapper classes or nested objects, extracting source specifications automatically. Second, building on taint-style vulnerabilities, we expand static taint analysis to track P2Xi-specific taint propagation by incorporating 5 categories of sensitive sink functions, covering over 236 specific sink implementations. Finally, to address false positives, we use LLMs to prune non-actionable sources and invalid taint propagation paths. For example, sources are pruned if the prompt passed to LLM APIs is hardcoded, as only user-controlled inputs (e.g., web inputs or function parameters) are considered valid taint sources. Additionally, we perform semantic analysis of sanitizers along the taint propagation path, identifying cases where data has been adequately validated or sanitized, and pruning these paths from further analysis. Together, these components enable TaintP2X to systematically identify P2Xi vulnerabilities with improved precision and minimal false positives.

Evaluation. To evaluate the effectiveness of TaintP2X, we constructed a ground truth dataset consisting of 35 known P2Xi vulnerabilities, spanning various categories such as code/command injection, SQL injection, arbitrary file read/write, and SSRF. TaintP2X successfully identified 27 of these vulnerabilities, achieving a recall

[1]Here, X serves as a wildcard, representing any sensitive operation or consequence that can result from the injection.

rate of 77.1%. In contrast, LLMSmith detected only 17 vulnerabilities, while AgentFuzz identified 7. Notably, both LLMSmith and AgentFuzz failed to automatically construct and execute the testing environments for a substantial portion of cases, which limited their detection capabilities.

To further assess the prevalence of P2Xi vulnerabilities in real-world software, we analyzed 4,772 open-source repositories collected from GitHub. TaintP2X reported a total of 101 taint propagation paths across 75 projects, including 74 instances of code/command injection, 15 instances of arbitrary file read/write, 1 instance of SQL injection, 9 instances of SSRF, and 2 instances of email injection. Among these, 7 vulnerabilities have been confirmed by developers, and 5 of them have been fixed until now. These results demonstrate the practical utility of TaintP2X in detecting P2Xi vulnerabilities at scale and its potential for enhancing the security of large-scale LLM-integrated ecosystems.

Contributions. In summary, we make the following contributions:

- **Novel Vulnerability Pattern.** We introduce and define a novel vulnerability paradigm called *Prompt-to-Anything Injection (P2Xi)*, which generalizes prior prompt injection vulnerabilities to encompass a wide range of security-sensitive operations, highlighting its risks in LLM-integrated applications.
- **Practical Tool.** We propose TaintP2X, a novel static taint analysis framework designed to detect P2Xi vulnerabilities. TaintP2X models LLM-generated outputs as taint source specifications, extends taint propagation analysis, and leverages LLM-assisted analysis to prune false positives, enabling precise and scalable detection of P2Xi vulnerabilities.
- **Large-scale Analysis in the Wild.** To assess the real-world impact of P2Xi vulnerabilities, we analyzed 4,772 open-source repositories and identified 101 taint paths across 75 repositories, with 7 vulnerabilities confirmed by developers, and 5 of them fixed. These findings underscore the prevalence of P2Xi vulnerabilities in LLM-integrated applications.

Artifact Availability. The full source code of TaintP2X is available at https://github.com/security-pride/TaintP2X.

2 BACKGROUND

2.1 LLM-integrated Applications

LLM-integrated applications, also called LLM agents, are software systems that integrate LLMs to enable intelligent and interactive functionalities, as illustrated in Figure 1. Typically, these applications adopt a modular architecture to support a variety of tasks, such as code generation, database querying, and web data scraping. The general workflow begins with a user submitting a natural language request through a user interface (UI). The request is then processed by a prompt template module, which builds an appropriate prompt and sends it to the LLM. The LLM generates executable actions or code based on the prompt, which is then executed by an action execution engine, which may involve database or network operations. The generated data is then processed and organized, and finally returned to the user interface for display. Throughout the process, the complexity of interacting with the LLM is seamlessly managed in the background, allowing users to access advanced features through a simple interface. This design enables developers

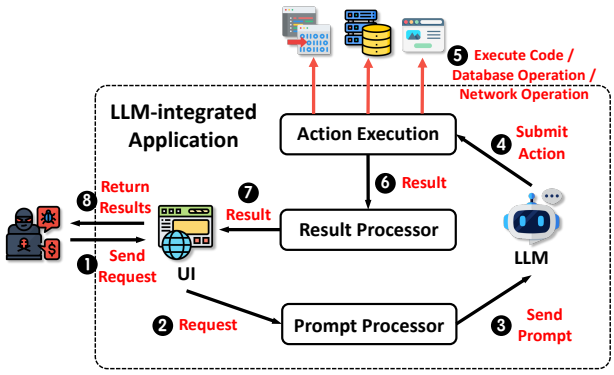

Figure 1: Workflow of LLM-integrated Applications.

to create highly customizable and responsive applications to meet a wide range of user needs.

2.2 Prompt-to-Anything Injection (P2Xi)

While LLM-integrated applications greatly extend the capabilities of modern software systems, integration with them also introduces new security challenges. In particular, implicit trust in model-generated content has given rise to a new class of vulnerabilities. A prominent example is "Prompt-to-Anything Injection" (P2Xi), a newly discovered vulnerability paradigm. P2Xi occurs when attacker-controlled inputs (directly or indirectly affecting prompts sent to LLM) result in the generation of malicious outputs that are subsequently executed or interpreted by downstream components. In this case, the prompts act as a conduit for injection-based attacks that threaten the integrity, confidentiality, or availability of the system. The "X" in P2Xi emphasizes the arbitrary nature of these downstream effects, which may include code execution, file operations, network requests, data exfiltration, or other unintended actions. This vulnerability stems from a common pattern in LLM-integrated applications: model outputs are frequently used not only for user interaction, but also as input to APIs, command-line tools, configuration files, or other automated workflows. When these outputs are automatically trusted and executed without adequate validation, sanitization, or isolation, maliciously crafted prompts can induce arbitrary and potentially dangerous behavior in the target system. Unlike traditional injection attacks (e.g., command injection or SQL injection), P2Xi exploits the generative and semantic properties of LLM, which makes detection by static analysis or simple pattern matching more difficult.

2.3 Problem Statement

Problem Overview. In many LLM-integrated applications, user inputs are dynamically embedded into prompt templates, and the resulting outputs are directly routed to sensitive components, such as code execution engines, database interfaces, or file system APIs. While this architecture significantly enhances automation and interactivity, it also introduces critical security risks. Since LLM outputs are not inherently trustworthy, any downstream execution of unvalidated model responses can lead to severe vulnerabilities, including

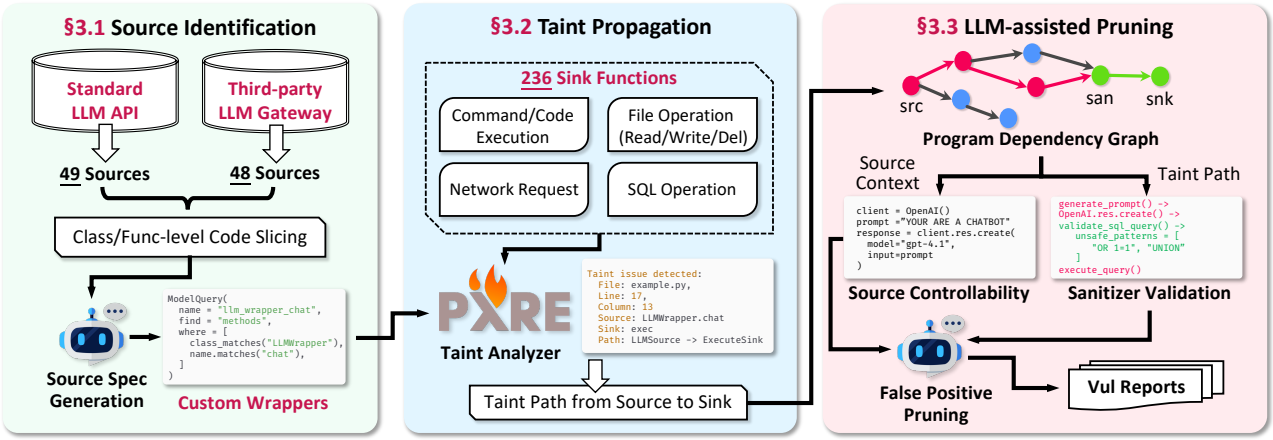

Figure 2: Overall Workflow of TaintP2X.

RCE, SQL injection, arbitrary file access, XSS, and SSRF. Moreover, the rich variety of API interfaces used in modern applications broadens the attack surface. Vulnerable LLM modules, once integrated into higher-level systems, can propagate risks throughout the software stack, thereby amplifying their impact.

Threat Model. In our threat model, we assume that the LLM-integrated application and its developers are benign and that the application's runtime environment has not been compromised. We consider the attacker to be a malicious user who is able to interact with the agent through regular operational channels. The main threat comes from prompt injection attacks, where the attacker writes prompts that trick the LLM into generating and executing harmful instructions through vulnerable APIs. We focus on two types of attackers: remote attackers and local attackers. **Remote attackers** interact with the agent through network-exposed interfaces such as web APIs or chatbot endpoints. By submitting crafted malicious prompts, they can trick the model into generating instructions that, when passed to vulnerable APIs, trigger server-side attacks. Such attacks may include RCE, SQL injection, XSS, arbitrary file read and write, or SSRF. This attack vector is common in cloud or server-hosted LLM agent services, as the APIs of these services are accessible over the Internet. **Local attackers** operate on the same device or local network as the agent. They may embed malicious prompts into files, documents, or other resources that the agent is configured to process or retrieve, such as files read from disk, content obtained from local or network drives, or external retrieval enhancement sources. When the agent obtains these resources, the embedded prompts may trigger sensitive actions or privilege escalation, which may lead to unauthorized actions or data leakage.

3 DESIGN OF TAINTP2X

To address the emerging challenges of detecting *P2Xi* vulnerabilities in LLM-integrated applications, we propose a novel static taint analysis framework, TaintP2X, which leverages both static analysis techniques and the reasoning capabilities of LLMs. The methodology of TaintP2X is designed to systematically model LLM-generated outputs as taint sources, track their propagation through program dependency graphs, and identify potential vulnerabilities in sensitive sink functions. Additionally, we incorporate LLM-assisted pruning mechanisms to improve precision by reducing false positives (FPs). As shown in Figure 2, TaintP2X consists of three key components: (1) *Source Identification*, where taint sources are extracted and modeled based on predefined specifications and dynamic analysis of wrapper functions; (2) *Taint Propagation Analysis*, which tracks the flow of tainted data through the program using static taint analysis techniques; and (3) *LLM-Assisted Pruning*, where false positives are mitigated through source controllability validation and sanitizer validation. Together, these components enable TaintP2X to detect P2Xi vulnerabilities with high accuracy and scalability.

3.1 Taint Source Identification

In this subsection, we describe how TaintP2X models taint sources through a combination of *Pre-defined Sources*, which include standard API interfaces and third-party LLM gateways, and *LLM-assisted Source Identification*, where LLMs are used to analyze custom or developer-defined code to extract taint source specifications.

3.1.1 Pre-defined Source Collection. To establish a comprehensive foundation for identifying taint sources, we curated a list of predefined sources by systematically analyzing the APIs and libraries provided by major LLM providers and widely used third-party frameworks. Specifically, we focused on two categories: (1) *Standard LLM APIs*, which include publicly available APIs from leading LLM providers such as OpenAI, Hugging Face, and Anthropic, and (2) *Third-party LLM Gateways*, which encompass middleware libraries like `LangChain` and `litellm` that wrap standard APIs for enhanced functionality or easier integration. The collection process for the pre-defined sources was heuristic in nature. Specifically, the first two authors of this paper independently conducted an extensive search to identify widely used APIs and libraries provided by prominent LLM vendors and frameworks. Their search spanned academic papers, technical reports, and active open-source repositories in both research and industrial contexts. After compiling the

Table 1: Examples of Pre-defined Taint Sources in TaintP2X.

Category	Provider	Example API/Method
Standard	OpenAI	`openai.ChatCompletion.create()`
	gemini	`google.generativeai.generate_text()`
	Anthropic	`anthropic.Anthropic.messages.create()`
	mistralai	`mistralai.client.MistralClient.chat()`
	groq	`groq.Groq.chat.completions.create()`
Third-party	LangChain	`langchain.chains.base.Chain.invoke()`
	llama_index	`llama_index.llms.openai.OpenAI.chat()`
	metagpt	`metagpt.llm.LLM.chat()`
	babyagi	`babyagi.BabyAGI.run()`
	litellm	`litellm.completion()`

initial lists, the two authors systematically reviewed the documentation to refine their respective lists by consolidating redundant entries, resolving ambiguities, and verifying the functionality of the APIs and libraries. Once each author had finalized their individual list, the two lists were merged into a single, unified collection. During the consolidation process, any discrepancies or disagreements regarding the inclusion of specific APIs or libraries were addressed through collaborative discussions. In cases where consensus could not be reached, the third author of this paper was brought in to mediate and provide a final decision. While this collection process does not guarantee exhaustive coverage due to the rapidly evolving ecosystem of LLM tools and libraries, we made a deliberate effort to include the most representative and widely used APIs. As shown in Table 1, we identified 49 standard API interfaces from major LLM providers and 48 third-party gateway methods across libraries such as `LangChain` and `litellm`, resulting in a total of 97 pre-defined sources (The full list is available in our open-source artifacts [30]). This ensures that TaintP2X is equipped to handle a diverse range of real-world applications, while also allowing for future extensibility as new APIs and libraries emerge.

3.1.2 LLM-assisted Source Identification. While pre-defined sources provide an initial foundation for identifying common taint sources, real-world applications often employ custom implementations, wrappers, or nested function calls that are not directly covered by standard APIs or third-party libraries. To address these scenarios, TaintP2X leverages LLM to assist in identifying additional taint sources dynamically, enabling the detection of vulnerabilities in developer-defined or less-documented codebases. The process consists of three key steps:

Parsing-based Pre-defined Source Identification. The first step in TaintP2X's LLM-assisted source identification process is to locate the usage of pre-defined sources within the codebase. This is achieved through parsing-based techniques that analyze the program's abstract syntax tree (AST). The goal is to systematically identify where pre-defined sources—such as API calls to `openai.ChatCompletion.create()` or `litellm.chat()`—are invoked. For each identified invocation, TaintP2X records its location in the code (e.g., class, method, or function) and its associated parameters. Parsing-based methods ensure precise matching by accounting for both direct invocations and indirect calls through aliases or imports. For example, if a developer uses `import litellm as llm`, the parser resolves `llm.chat()` to the corresponding pre-defined source. This step establishes a robust mapping between pre-defined sources and their occurrences, serving as the entry point for further analysis.

System

You are a static analysis expert. Analyze the provided code snippet and determine if it acts as a taint source.
TASK:
- Check if the function or method interacts with LLM APIs or gateways.
- Determine if the data passed to the taint source is user-controlled.
- If applicable, classify the function/method as a taint source and output a JSON result.
INPUT FORMAT: — [Code snippet] —
OUTPUT FORMAT:
Return your result in the following JSON format:

```
{
    "method_name": "<Method Name>",
    "is_llm_call": true/false,
    "description": "<Explanation of the analysis result>"
}
```

Figure 3: Simplified Prompt Template for LLM-assisted Taint Source Identification.

Class/Function-Level Code Slicing. Once pre-defined sources are located, TaintP2X performs class/function-level code slicing to extract the surrounding code context. This step involves isolating the method or class where the pre-defined source is invoked, along with its dependencies, control flow, and data flow. The slicing process ensures that all relevant code contributing to the behavior of the pre-defined source is included. For instance, if a pre-defined source such as `litellm.chat()` is called within a wrapper method `LLMWrapper.chat()`, TaintP2X extracts the entire method implementation, including its parameters, return values, and any intermediate operations. Similarly, if the invocation is part of a class, the relevant attributes, constructors, and methods are included to provide a complete picture of the code context. This step ensures that the extracted code captures critical information needed for taint source modeling, such as how input data flows into and out of the pre-defined source.

Source Specification Generation. After obtaining the sliced code context, TaintP2X leverages LLMs to generate structured taint source specifications. The extracted code is provided as input to the LLM, along with a carefully designed prompt to guide its reasoning, as illustrated in Figure 3. The prompt explicitly asks the LLM to analyze whether the function or method serves as a taint source and, if so, to output a structured source specification in JSON format. This step enables TaintP2X to dynamically augment its list of taint sources by incorporating developer-defined wrappers and custom implementations that interact with pre-defined sources. By leveraging the semantic reasoning capabilities of LLMs, TaintP2X ensures accurate identification of taint sources, even in complex or undocumented codebases.

3.2 Taint Propagation

To comprehensively and efficiently detect taint propagation paths, we divide the process into three key components: intra-procedural propagation, inter-procedural taint derivation, and intersection analysis. Each component focuses on a specific aspect of taint propagation, ensuring precise tracking of taint flows across program variables and procedures.

Intra-Procedural Data Flow Construction. The first phase of the analysis focuses on constructing a complete representation of data flow relationships within individual procedures. Each procedure p is modeled as a control-flow graph (CFG), denoted as $\text{CFG}_p = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ represents the set of program statements (nodes) and $\mathcal{E}$ represents the control-flow edges. The goal of this phase is to identify all data dependencies within the procedure, capturing how variables are defined, used, and propagated across different program points. To achieve this, the analysis tracks the flow of data through assignments, computations, and control structures in the procedure. When a variable v is defined or updated at a statement s_1 and subsequently used at another statement s_2, a data flow edge is established between s_1 and s_2. Such edges collectively form a data flow graph for the procedure, which captures all relevant dependencies between variables, ensuring that downstream analyses have a complete foundation for reasoning about potential taint propagation.

Inter-Procedural Taint Derivation. After intra-procedural propagation, the analysis extends across function boundaries using the program's call graph $CG = (F, C)$, where F is the set of functions, and $C \subseteq F \times F$ represents the call relationships between them. Inter-procedural propagation is performed in two complementary directions: forward propagation from taint sources and backward propagation from taint sinks. In forward propagation, if a function f is identified as a taint source and its return value is used as an argument in a call to another function g, the taint is propagated to g. For example, if g contains a statement $y = f(x)$, and f is a taint source, then y is tainted in g, and g itself may be marked as a derived taint source. Conversely, in backward propagation, the analysis starts from functions identified as taint sinks and traces whether their parameters are influenced by upstream functions. If a function g is a taint sink and is invoked from f with a tainted argument, the corresponding calling parameter in f is marked as sink-reachable. This backward propagation ensures that all functions in the call chain contributing to the taint sink are identified. By iteratively propagating taint information along the call chain in both directions, the analysis constructs a comprehensive view of how data flows through the program.

Intersection Analysis. The final phase of the analysis synthesizes the results of forward and backward inter-procedural propagation to identify complete taint propagation paths. This is achieved by computing the intersection of the sets of functions influenced by taint sources and those contributing to taint sinks. Let $\mathcal{S}_f$ denote the set of functions reachable from taint sources during forward propagation, and $\mathcal{S}_b$ denote the set of functions identified as influencing taint sinks during backward propagation. The intersection $\mathcal{I} = \mathcal{S}_f \cap \mathcal{S}_b$ represents the set of functions that lie on both the source-to-sink propagation path and contribute to the overall taint flow. For each function $f \in \mathcal{I}$, the analysis verifies whether there exists a valid data flow path connecting a taint source to a taint sink through f. This involves examining both intra-procedural data flow graphs and inter-procedural taint dependencies to ensure that the propagated taint reaches the sink without interruption. Functions in $\mathcal{I}$ may serve as critical intermediaries in the taint flow, propagating taint from their input parameters to their return values or from their parameters to sensitive operations (sinks) within their body.

3.3 LLM-Assisted Pruning

To enhance the precision of static taint analysis and reduce FPs, TaintP2X integrates an *LLM-assisted pruning* mechanism that leverages the semantic reasoning capabilities of LLMs. The pruning process is divided into two core components: *source controllability analysis* and *sanitizer validation*.

3.3.1 Source Controllability Analysis. This component determines whether the content of prompts provided to LLMs is user-controllable, a critical factor in identifying prompt injection vulnerabilities. To achieve this, TaintP2X statically reconstructs the full execution context of each LLM invocation, capturing both local variables (e.g., function arguments and local assignments) and global context (e.g., configuration constants and module-level settings). Prompts are then classified as either user-controllable, if they are derived from user inputs, function parameters, or other dynamic sources, or non-user-controllable if they are solely constructed from hardcoded string literals or immutable constants. User-controllable prompts are flagged for further analysis, while non-user-controllable prompts are excluded to reduce unnecessary false positives.

3.3.2 Sanitizer Validation. The second component focuses on validating whether taint propagation chains identified during static analysis include effective sanitization mechanisms. This step addresses the challenge of distinguishing between functions that genuinely sanitize inputs and those that merely resemble sanitizers in naming or structure. TaintP2X employs a multi-turn, interactive validation framework powered by LLMs to evaluate each suspicious taint chain. The process begins with decomposing the taint chain into function-level sub-tasks, where the LLM is guided by customized prompts that incorporate prior knowledge such as exploit patterns, known evasion techniques, and common false-positive scenarios. The LLM evaluates whether the taint source is valid, whether sanitization functions within the chain effectively neutralize taint, and whether the chain as a whole constitutes an exploitable vulnerability. To enhance robustness, TaintP2X employs a recursive prompt construction strategy for intermediate nodes in the taint chain, enabling the LLM to inherit upstream context while leveraging auxiliary tools (e.g., static code search) to recover surrounding semantics.

4 EVALUATION

To systematically evaluate the effectiveness of TaintP2X, we focus on the following research questions (RQs):

RQ1: How effective is TaintP2X in detecting known P2Xi vulnerabilities? This research question assesses the accuracy and completeness of TaintP2X on a curated dataset of real-world P2Xi vulnerabilities.

Table 2: Comparison Results with LLMSmith and AgentFuzz.

Type	Repository & Version	CVE ID	LLMSMITH	AgentFuzz	TaintP2X
Code Injection	langchain 0.0.131	CVE-2023-29374	Y	-	Y
	langchain 0.0.194	CVE-2023-36095	Y	-	Y
	langchain 0.0.232	CVE-2023-39659	Y	-	N
	langchain <0.0.236	CVE-2023-36258	Y	-	Y
	langchain 0.0.64	CVE-2023-36188	Y	-	Y
	langchain 0.0.194	CVE-2023-38896	Y	-	Y
	langchain 0.0.231	CVE-2023-38860	Y	-	Y
	langchain 0.0.245	CVE-2023-39631	N	-	Y
	langchain <0.0.306	CVE-2023-44467	Y	-	Y
	langchain <0.1.8	CVE-2024-27444	Y	-	Y
	langchain_experimental <0.0.61	CVE-2024-38459	Y	-	N
	litellm 1.40.12	CVE-2024-6825	N	-	N
	llama_index 0.10.25	CVE-2024-3271	Y	-	Y
	llama_index 0.9.46	CVE-2024-3098	Y	-	Y
	llama_index 0.7.13	CVE-2023-39662	Y	-	Y
	MetaGPT v0.6.3	CVE-2024-23750	N	-	Y
	OpenManus 2025.3.13	CVE-2025-2733	N	N	Y
	PandasAI 0.8.0	CVE-2023-39660	Y	-	Y
	PandasAI 0.8.1	CVE-2023-39661	Y	-	Y
	vanna 0.3.3	CVE-2024-5826	Y	Y	Y
	Vanna 0.3.1	CVE-2024-5565	Y	Y	Y
	autogpt v0.5.0	CVE-2024-1881	N	Y	N
SQL Injection	langchain <v0.0.247	CVE-2025-27135	-	-	Y
	langchain 0.2.5	CVE-2024-8309	-	-	N
	llama_index 0.11.23	CVE-2024-12911	-	-	Y
	llama_index 0.9.28.post2	CVE-2024-23751	-	-	N
	vanna 0.3.4	CVE-2024-5753	-	Y	Y
	vanna 0.3.4	CVE-2024-5827	-	Y	Y
	Vanna 0.6.2	CVE-2024-7764	-	Y	N
File Read/Write	Superagi v0.0.14	-	-	N	Y
	devika	CVE-2024-5927	-	N	Y
	devika	CVE-2024-5821	-	N	Y
	devika	CVE-2024-6331	-	N	Y
SSRF	quivr 0.0.236	CVE-2024-5885	-	Y	Y
	langchain 0.0.327	CVE-2023-32786	-	-	N
Total Detected			**17**	**7**	**27**

Note. *Superagi v0.0.14* is included based on a public vulnerability disclosure on huntr and has not yet been assigned a CVE. A dash (-) indicates that the corresponding tool is unable to analyze the given repository, either due to unsupported frameworks, language mismatches, or insufficient coverage of the vulnerability type.

RQ2: What is the contribution of each core module in TaintP2X to the overall detection performance? This question conducts an ablation study to evaluate the individual impact of TaintP2X's four major components, including taint source identification, static taint propagation, prompt controllability analysis, and LLM-based validation.

RQ3: How well does TaintP2X perform in analyzing large-scale, real-world LLM-integrated applications? This question examines the scalability and practical effectiveness of TaintP2X across a wide range of open-source projects drawn from the real-world LLM application ecosystem.

4.1 Evaluation Setup

Tool Implementation. We implemented a prototype of TaintP2X based on the open-source static analysis framework Pysa, comprising over 3,500 lines of Python code (excluding third-party libraries and external tools). The source identification module combines static parsing with semantic reasoning, leveraging AST analysis and LLM-generated structured specifications to automatically identify both standard and custom taint sources. The taint propagation module, built on Pysa, constructs dataflow graphs, performs inter-procedural propagation, and applies intersection analysis to accurately trace complete source-to-sink propagation paths. Finally, the LLM-assisted pruning module employs a semantics-driven two-stage process that integrates source controllability analysis with sanitizer validation, effectively eliminating FPs.

Environment. All experiments were conducted on a server equipped with two AMD EPYC 7713 processors (128 cores) and 512 GB of RAM, running Ubuntu 22.04.5 LTS. For LLM-based analysis and verification, we employed the DeepSeek-V3 model via its API, allowing the system to interact with the model for structured taint source generation and taint propagation validation.

Baseline. For baseline selection, we selected two SOTA systems (LLMSmith [21] and AgentFuzz [8]) as baselines for P2Xi vulnerability detection. LLMSmith combines lightweight static analysis with white-box and black-box testing techniques to detect LLM4Shell-style vulnerabilities, while AgentFuzz builds upon this

by introducing guided grey-box fuzzing specifically targeting taint-style vulnerabilities in LLM-based agent systems.

Dataset. For the ground truth dataset used in RQ1, we have collected 35 real-world P2Xi vulnerabilities. Specifically, we gathered 17 publicly disclosed vulnerabilities in total from LLMSmith and AgentFuzz. In addition, two authors independently searched for relevant P2Xi vulnerabilities using keywords such as "prompt injection" in bug bounty platforms like huntr [13] and vulnerability databases like NVD [26] and GitHub Advisory Database [9]. After merging the results and removing duplicates, we obtained an additional 18 vulnerabilities. For the large-scale evaluation in RQ3, we developed a crawler that collected 41,266 open-source GitHub repositories containing both LLM client instantiations and sensitive sink functions, from which we selected 4,772 high-quality repositories (each with at least 50 stars) to build a comprehensive dataset of real-world LLM-integrated applications.

4.2 RQ1: Effectiveness of TaintP2X

We evaluate the detection effectiveness of TaintP2X using our curated ground-truth dataset of 35 P2Xi vulnerabilities. Each vulnerability is manually verified and linked to a publicly disclosed CVE identifier or a well-documented community report to ensure the authenticity and practical relevance of the dataset. These cases cover four key categories commonly found in LLM-integrated applications: code injection, SQL injection, arbitrary file access, and SSRF. All samples are from widely adopted open-source LLM projects to maximize representativeness.

We applied TaintP2X to this benchmark and evaluated its performance using standard detection metrics, including **true positives (TP)**, **false negatives (FN)**, and overall **recall**. As described in Table 2, TaintP2X successfully detected 27 out of 35 vulnerabilities with a recall rate of **77.1%**. This high recall demonstrates that TaintP2X is capable of identifying complex Prompt-to-Execution attack paths by leveraging its combination of static taint tracking and LLM-based semantic validation. It is able to reason about deep, multi-stage logic and contextual semantics to discover subtle vulnerabilities that are usually not discovered by traditional analysis tools. For comparison, we selected two state-of-the-art detection systems as baselines: LLMSMITH and AgentFuzz. LLMSMITH relies on static template matching and detected 17 vulnerabilities but failed to identify those involving SQL injection, file operations, or SSRF. AgentFuzz, which focuses on dynamic analysis in LLM applications, lacks support for internal LLM framework instrumentation and detected only 7 vulnerabilities. Beyond overall recall, we further analyzed performance across different vulnerability categories (see Table 2). Results show that TaintP2X maintains robust detection capabilities across all four categories, with particularly strong performance in detecting file-related and SSRF vulnerabilities, attributed to its fine-grained modeling of LLM output semantics and controllable input propagation.

4.3 RQ2: Ablation Study

To systematically assess the contribution of each core module to overall vulnerability detection, we conducted an ablation study on a large-scale dataset of 4,772 real-world LLM-integrated applications. The results are shown in Table 3. We first evaluated the

Table 3: Ablation Study Results Across TaintP2X Variants.

Modules	TP	FP	P	TN	FN	N	Total
M2	79	139	218	–	–	4,554	4,772
M1+M2	95	168	263	–	–	4,509	4,772
M2+M4	69	42	111	107	0	107	218
M1+M2+M4	85	47	132	131	0	131	263
M1+M2+M3+M4	78	36	114	145	4	149	263

Note. TP = True Positives, FP = False Positives, P = Total Positives (TP+FP), TN = True Negatives, FN = False Negatives, N = Total Negatives (TN+FN). M1: Taint Source Identification Module. M2: Static Taint Propagation Analysis Module. M3: Prompt Controllability Analysis Module. M4: LLM-based Risk Chain Validation Module.

static taint propagation analysis module (M2) alone, identifying 218 items that may have taint propagation paths and forming the initial candidate set (**P**). At this stage, the system obtained 79 TPs and 139 false positives (FPs), indicating that M2 has a high recall rate in large-scale static analysis. However, due to the lack of semantic verification, the FP rate is relatively high. Next, by integrating the **taint source identification module (M1)**, the TaintP2X is able to identify developer-defined wrapper classes and nested objects as taint sources, thereby enhancing the completeness of the propagation graph. The combined configuration of M1+M2 expands the candidate set to 263 items, increases the TPs to 95, and increases the FPs to 168, thereby expanding the coverage of potential vulnerabilities.

To mitigate FPs introduced by static analysis, we integrated the **LLM-based risk chain validation module (M4)**, which supports context-aware semantic verification of candidate call chains. Under the M2+M4 configuration, 111 call chains out of 218 M2 detected paths were verified, 69 TPs and 42 FPs were confirmed, and the detection accuracy was significantly improved from 36.2% to 62.2%. Building on this, the M1+M2+M4 combination screened out 132 high-confidence call chains from 263 candidate call chains, of which 85 confirmed TPs and 47 FPs, effectively balancing comprehensive coverage and improved accuracy. Finally, we introduced the **prompt controllability analysis module (M3)** to perform fine-grained modeling of user input controllability. Under the full configuration (M1+M2+M3+M4), TaintP2X further improved the discrimination of negative samples, and the TN increased from 131 to 145. A small number of FNs (FN=4) were observed, which was mainly attributed to the stricter semantic modeling introduced by the prompt controllability analysis.

Overall, M4 significantly enhanced both precision and negative sample discrimination, while M3 contributed to a balanced optimization between precision and recall by better capturing the interaction between user input and sensitive operations. This underscores the importance of M3 in identifying the practical impact of controllable prompts on exploitability, particularly in complex Prompt-to-Execution attack scenarios, and in improving the system's semantic understanding of potential threats.

4.4 RQ3: Real-World Applicability

To answer RQ3, we designed an empirical study to comprehensively evaluate the practical effectiveness and deployment adaptability

of TaintP2X in real LLM application scenarios. Our evaluation focuses on two perspectives:

- **RQ3.1: End-to-End System Performance Evaluation.** We assessed the overall performance of the complete system configuration (M1+M2+M3+M4) on vulnerability detection tasks, as detailed in Table 3.
- **RQ3.2: Vulnerability Type Coverage and Community Feedback.** We analyzed the tool's detection capabilities across representative vulnerability types, as well as its actual acceptance within the open-source community.

4.4.1 RQ3.1: End-to-End System Performance Evaluation. With the complete system configuration enabled, TaintP2X successfully detected 78 TPs, 36 FPs, 145 TNs, and only 4 FNs, demonstrating strong precision and recall. This strong performance is attributed to the tool's accurate modeling of user input controllability and its context-aware semantic validation of sensitive call chains. As a result, TaintP2X is able to identify complex prompt-to-execution attack paths that are often missed by traditional detection methods. To better understand the causes of FPs, we conducted a systematic analysis of erroneous cases and identified three main reasons: (1) Taint source misidentification, e.g., some functions syntactically similar to LLM APIs were mistakenly marked as taint sources; (2) Over-approximation of sinks, such as `subprocess.Popen` with `shell=False`, were flagged even though they are not actually exploitable; (3) Omission of runtime mitigation mechanisms, where some static paths are blocked by runtime path validation or whitelisting. These findings indicate that while TaintP2X maintains high recall, it also effectively controls FPs, underscoring its strong engineering practicality.

To account for the hallucination and uncertainty often associated with LLMs, we further conducted a comprehensive evaluation of the robustness of LLM-assisted analysis. Specifically, we conducted an experiment involving a manual audit of 100 randomly sampled taint paths identified and pruned by the LLM-assisted modules (M1 and M4). Three independent security experts reviewed the results to establish ground-truth benchmarks for validation. The inter-rater agreement between the human auditors and the LLM outputs, measured using Cohen's κ coefficient, reached 0.85, indicating a high level of consistency and reliability in identifying valid taint paths and pruning FPs. Furthermore, we performed additional experiments to assess the robustness of TaintP2X under different LLM architectures and prompting strategies. Using OpenAI's GPT-3.5, GPT-4, and Anthropic's Claude, we observed module-level consistency exceeding 96% for M1 and 91% for M4, demonstrating that the LLM-assisted components yield stable and reproducible results across distinct model architectures. Regarding prompting strategies, we evaluated zero-shot, few-shot, and chain-of-thought (CoT) prompting variants, achieving an overall consistency of 94.12%, thereby confirming the robustness of the LLM-assisted analysis.

4.4.2 RQ3.2: Vulnerability Type Coverage and Community Feedback. Regarding vulnerability type coverage, TaintP2X supports detection of multiple critical vulnerability classes, including code injection (e.g., `exec`, `subprocess.run`), arbitrary file access, SSRF, and SQL injection. Empirical analysis shows a significantly higher number of sinks related to code injection issues, reflecting the widespread exposure of such risks in contemporary large model applications, as presented in Table 4. To verify the exploitability of potential vulnerabilities, we selected high-confidence paths generated by the system and submitted them to relevant open-source developers. Among the 40 submitted issues, 7 were confirmed by maintainers, and some projects have initiated remediation efforts. Unsubmitted issues were mainly due to existing security advisories or documentation, low-impact vulnerabilities (e.g., access to non-sensitive local files), or vulnerabilities located within test or demonstration code not affecting production environments.

Table 4: Sink Function Calls by Vulnerability Type.

Vulnerability Type	Sink Function	# Calls
Code Injection	`exec`	31
	`eval`	19
	`subprocess.run`	16
	`os.system`	3
	`subprocess.call`	2
	`subprocess.Popen.__init__`	1
	`importlib.import_module`	2
SQL Injection	`sqlite_utils.Database.query`	1
File Operations	`open`	15
SSRF	`requests.api.get`	9
Data Leakage	`email.message.Message.attach`	2

4.4.3 Case Study#1: Prompt-to-Code-Injection in ComfyUI. To further understand the practical risk of prompt-to-code injection in real-world LLM-integrated systems, we first perform a case study on a vulnerability in the `AnyNode` component of `ComfyUI`. This module allows end users to dynamically execute LLM-generated Python code via a visual workflow. As illustrated in Listing 1, the internal execution flow translates user-provided prompts into Python scripts, which are then executed within the host environment. Although a blacklist-based `CodeSanitizer` is applied during the `safe_exec` stage to filter prohibited APIs (e.g., `eval`, `exec`, `input`, `open`), its coverage is incomplete. As the code snippet shows, the LLM response (`r`) is directly incorporated into `self.script` after extraction, and the sanitized script is eventually executed via `exec`. Because only a narrow set of APIs is covered by the blacklist and key primitives (e.g., `io.FileIO`) are not included, an adversarial prompt can steer the LLM to produce code that is semantically dangerous yet syntactically compliant with the sanitizer's rules. In our proof-of-concept, the generated script used alternative filesystem APIs to create a file `/pwn` containing the string `hacked`, confirming that an attacker can gain write access to the underlying file system. This case demonstrates that blacklist-based sanitization is insufficient in protecting LLM-integrated systems. A more robust design would replace blacklist filtering with a conservative, whitelist-based policy or execute LLM-generated code inside a constrained sandbox.

4.4.4 Case Study#2: Prompt-to-File-Write in `gpt-engineer`. We further conduct a case study on a prompt-to-file-write vulnerability in `gpt-engineer`, a framework that enables LLMs to modify local

Listing 1: Prompt-to-Code-Injection in ComfyUI.

```
class CodeSanitizer(ast.NodeVisitor):
    def __init__(self):
        self.dangerous_constructs = [
            'eval', 'exec', 'input', '__import__', 'os',
                'subprocess', 'shutil', 'sys', 'compile'
        ]

def go(self, prompt, ...):
    if need_generation(prompt):
        final_template = self.render_template(
             SYSTEM_TEMPLATE, ...)
        r = self.get_llm_response(prompt, ...)
        self.script = self.extract_code_block(r)

    self.safe_exec(modified_script, globals_dict,
         locals_dict)

    exec(sanitize_code(code_string), globals_dict,
         locals_dict)

def sanitize_code(code):
    tree = ast.parse(code)
    sanitizer = CodeSanitizer()
    sanitizer.visit(tree)
```

Listing 2: Prompt-to-File-Write in gpt-engineer

```
goal = get_user_goal()

files = llm_select_files_and_instructions(goal)

for file in files:
    path = ensure_prefix("/tmp/repo/", file["path"])

    content = read_file(path)  # open(path, "r")

    resp = llm_edit_code(messages=build_messages(content,
          goal),functions=["edit_repo_file"])

    if resp.function_call.name == "edit_repo_file":
        changes = json_loads(resp.function_call.arguments
             )["changes"]
        apply_line_changes(path, changes)
        # open(path, "w")
    else:
        log_error("No function call found")
```

codebases autonomously. As shown in Listing 2, it takes a natural-language goal, lets the LLM select target files and edits, and then applies these edits directly to the filesystem. Although `gpt-engineer` attempts to constrain modifications to the `/tmp/repo/` workspace through path prefixing, this mechanism is applied before path normalization and therefore fails to sanitize path traversal attacks. As a result, a prompt can induce the LLM to output paths such as `/tmp/repo/../../root/hacked.txt` that pass the prefix check but, after normalization by the underlying OS, resolve outside the designated workspace. In our proof-of-concept, this allowed an LLM-generated function call to create `/root/hacked.txt` containing the string `hacked`, demonstrating a prompt-to-file-write primitive that escapes the intended file boundary. This case reveals that robust protection in LLM-integrated applications requires resolving paths to a canonical form, enforcing workspace membership on the normalized path, and introducing human-in-the-loop approval for LLM-suggested file operations.

5 DISCUSSION

Mitigation. Recent advances in mitigating P2Xi vulnerabilities within LLM-based applications can be broadly divided into three main approaches. First, confirmation mechanisms require explicit human approval before executing sensitive actions [44]. Second, input/output filtering systems attempt to detect and block malicious content during the prompt construction and model output phases [23]. Third, tool isolation strategies restrict the LLM to interacting only with a predefined set of tools, while a system-level controller disables all non-whitelisted capabilities [6]. Although these strategies establish a basic level of security, each presents significant limitations. Confirmation mechanisms often undermine system automation and usability, and fatigued or inattentive reviewers may inadvertently approve unsafe actions. Filter-based defenses are generally heuristic in nature, and thus cannot guarantee comprehensive coverage against adaptive or obfuscated attack patterns. Tool isolation, though more rigorous, introduces implementation complexity and does not eliminate the risk that even approved tools might be misused for malicious purposes. Moreover, we observe that a significant portion of developers have not implemented any security countermeasures at all. This highlights an urgent need to promote security-conscious development practices to ensure LLM applications are robust against P2Xi attacks.

Generalizability and Scalability. The design of TaintP2X is inherently language-agnostic and highly extensible. Since the core workflow only relies on taint propagation traces extracted from static analysis tools, the approach can be readily adapted to other programming languages, provided that suitable static analysis backends are available. Furthermore, the general capability of LLMs to semantically interpret multiple languages [32] enhances the system's applicability across diverse ecosystems. In addition, the current evaluation focuses on high-profile repositories with sufficient community attention (e.g., star thresholds), but the analysis framework can be naturally scaled to a broader corpus of LLM-powered applications. This includes systematic scanning of all public repositories on platforms such as GitHub, thereby enabling large-scale, ecosystem-wide assessment of P2Xi vulnerabilities.

Limitations. Despite the strong performance of TaintP2X in detecting P2Xi vulnerabilities, several limitations remain. Firstly, the detection capability is inherently limited by the underlying static analysis engine, which struggles to capture implicit taint propagation through mechanisms such as reflection, dynamic imports, or runtime code generation. These techniques are frequently encountered in real-world LLM applications [16]. Incorporating dynamic or hybrid analysis techniques in future work could help address these blind spots. Secondly, the empirical evaluation is conducted on a curated set of open-source projects with relatively high community visibility (e.g., selected by GitHub star count). This sampling strategy may exclude vulnerabilities present in long-tail or emerging projects. A broader, star-agnostic analysis across the GitHub ecosystem is necessary to obtain a more representative assessment of LLM-integrated application security. Thirdly, TaintP2X currently only supports Python-based projects. While Python remains the dominant language in the LLM application domain, extending support to additional languages, such as JavaScript, TypeScript, or Rust, would enhance the generalizability and practical utility

of the system. Finally, although TaintP2X achieves a high recall rate, the false positive rate remains relatively high. Future work could explore more advanced prompt engineering techniques [49], constraint propagation, or symbolic reasoning [35] to improve precision while maintaining broad vulnerability coverage.

6 RELATED WORK

With the integration of LLMs into key tasks such as natural language processing, code generation, and dialogue systems, their security risks have garnered increasing attention. Existing research primarily focuses on the following three areas:

Evaluating intrinsic model safety. For model evaluation, various high-coverage safety benchmarks have been proposed. SafetyBench [47] provides over 11,000 multiple-choice questions across seven risk categories for fine-grained robustness assessment. ALERT [37] constructs over 45,000 red-teaming prompts and incorporates a detailed risk taxonomy to simulate overreach and data leakage scenarios. Lightweight test suites such as SimpleSafetyTests [38] and context-adaptive alignment methods like SafeInfer [1] further strengthen practical safety evaluations. Cybersecurity-focused benchmarks, including CyberSecEval 2 [2], SECURE [3], and CS-Eval [46], expand the evaluation scope to include prompt injection, code interpreter abuse, and multimodal attack vectors.

Modeling and executing adversarial prompt attacks. In adversarial prompting, studies demonstrate that carefully crafted input sequences can induce unauthorized or harmful outputs without requiring access to model weights. Common attack types include prompt injection, contextual poisoning, and CoT hijacking. Representative works include indirect prompt injection by Greshake *et al.* [10], black-box jailbreaks across popular LLM applications [22], and cross-model transferable adversarial prompts [50].

Designing robust defense mechanisms. To mitigate such attacks, multi-level defense strategies have been proposed, including adversarial training, input purification (e.g., LLAMOS [19]), output screening (e.g., LLM Self Defense [29]), and ontology-driven assurance (e.g., [24]). Modal-specific frameworks like SpeechGuard [28] enhance robustness in voice-based input scenarios. Some works have also explored proactive countermeasures, exploiting the attack model's bias and memory limitations for misdirection.

Although significant progress has been made in model-level security evaluation and defense, there remains a gap in systematic detection methods for system-level vulnerabilities caused by prompt injection, such as RCE and SQL injection. LLMSmith[21] proposed a lightweight approach that combines static analysis with white-box/black-box testing to identify injection risks in framework source code. Building on this, AgentFuzz[8] introduced a targeted gray-box fuzzing technique to detect taint-style vulnerabilities in LLM-based agent systems. Pedro *et al.* [27] further revealed prompt-to-SQL injection (P2SQL) attack paths in frameworks like LangChain and LlamaIndex. However, existing methods are predominantly attack-specific and lack unified modeling across different prompt injection variants. Moreover, their reliance on dynamic analysis constrains scalability when applied to large codebases. TaintP2X addresses these limitations by providing a comprehensive static tool that generalizes across P2Xi vulnerabilities while maintaining practical scalability for real-world applications.

7 CONCLUSION

In this paper, we present TaintP2X, a static taint analysis framework for detecting P2Xi vulnerabilities in LLM-integrated applications. We formalize existing attack patterns into a unified paradigm, identifying implicit trust in LLM outputs as the fundamental vulnerability. TaintP2X extends conventional taint tracking by treating LLM-generated content as user-influenced sources and incorporates semantic validation to reduce FPs. Our evaluation on 35 known vulnerabilities and 4,772 GitHub repositories demonstrates TaintP2X's effectiveness, detecting 101 valid taint paths and revealing previously unknown vulnerabilities confirmed by developers. These findings highlight the widespread prevalence of P2Xi risks and underscore the need for automated security auditing in LLM-integrated applications.

ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China (grants No.62572209, 62502168) and the Hubei Provincial Key Research and Development Program (grant No. 2025BAB057).

REFERENCES

[1] Somnath Banerjee, Sayan Layek, Soham Tripathy, Shanu Kumar, Animesh Mukherjee, and Rima Hazra. 2024. SafeInfer: Context Adaptive Decoding Time Safety Alignment for Large Language Models. arXiv:2406.12274 [cs.CL] https://arxiv.org/abs/2406.12274

[2] Manish Bhatt, Sahana Chennabasappa, Yue Li, Cyrus Nikolaidis, Daniel Song, Shengye Wan, Faizan Ahmad, Cornelius Aschermann, Yaohui Chen, Dhaval Kapil, David Molnar, Spencer Whitman, and Joshua Saxe. 2024. CyberSecEval 2: A Wide-Ranging Cybersecurity Evaluation Suite for Large Language Models. arXiv:2404.13161 [cs.CR] https://arxiv.org/abs/2404.13161

[3] Dipkamal Bhusal, Md Tanvirul Alam, Le Nguyen, Ashim Mahara, Zachary Lightcap, Rodney Frazier, Romy Fieblinger, Grace Long Torales, Benjamin A. Blakely, and Nidhi Rastogi. 2024. SECURE: Benchmarking Large Language Models for Cybersecurity. In *2024 Annual Computer Security Applications Conference (ACSAC)*. 15–30. https://doi.org/10.1109/ACSAC63791.2024.00019

[4] The MITRE Corporation. 2018. CAPEC-23: File Content Injection. https://capec.mitre.org/data/definitions/23.html Accessed: 2025-07-09.

[5] CrewAI. 2025. Tools: CrewAI Documentation. https://docs.crewai.com/en/concepts/tools. Accessed: 2025-07-09.

[6] Edoardo Debenedetti, Jie Zhang, Mislav Balunović, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. 2024. AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents. arXiv:2406.13352 [cs.CR] https://arxiv.org/abs/2406.13352

[7] Hugging Face. 2023. Function Calling: Hugging Face Documentation. https://huggingface.co/docs/hugs/guides/function-calling#best-practices. Accessed: 2025-07-09.

[8] Liu Fengyu. 2025. *Make Agent Defeat Agent: Automatic Detection of Taint-Style Vulnerabilities in LLM-based Agents.* https://doi.org/10.5281/zenodo.15590097

[9] Github. 2025. Github Advisory Database. https://github.com/advisories. Accessed: 2025-07-09.

[10] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. arXiv:2302.12173 [cs.CR] https://arxiv.org/abs/2302.12173

[11] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* 33, 8, Article 220 (Dec. 2024), 79 pages. https://doi.org/10.1145/3695988

[12] Jie Huang and Kevin Chen-Chuan Chang. 2023. Towards Reasoning in Large Language Models: A Survey. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, 1049–1065. https://doi.org/10.18653/V1/2023.FINDINGS-ACL.67

[13] huntr. 2025. The world's first bug bounty platform for AI/ML. https://huntr.com/. Accessed: 2025-07-09.

[14] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From LLMs to LLM-based Agents for Software Engineering: A Survey of

Current, Challenges and Future. *CoRR* abs/2408.02479 (2024). https://doi.org/10.48550/ARXIV.2408.02479 arXiv:2408.02479

[15] Md. Monjurul Karim, Dong Hoang Van, Sangeen Khan, Qiang Qu, and Yaroslav Kholodov. 2025. AI Agents Meet Blockchain: A Survey on Secure and Scalable Collaboration for Multi-Agents. *Future Internet* 17, 2 (2025), 57. https://doi.org/10.3390/FI17020057

[16] Junhyoung Kim, TaeGuen Kim, and Eul Gyu Im. 2014. Survey of dynamic taint analysis. In *2014 4th IEEE International Conference on Network Infrastructure and Digital Content*. 269–272. https://doi.org/10.1109/ICNIDC.2014.7000307

[17] LangChain. 2025. Tool Calling: LangChain Documentation. https://python.langchain.com/docs/concepts/tool_calling/. Accessed: 2025-07-09.

[18] Zenan Li, Yuan Yao, Xiaoxing Ma, and Jian Lv. 2025. Neuro-symbolic systems: a perspective of uncertainty management. *SCIENTIA SINICA Informationis* 55, 1 (2025), 1–. https://doi.org/10.1360/SSI-2024-0163

[19] Guang Lin, Toshihisa Tanaka, and Qibin Zhao. 2025. Large Language Model Sentinel: LLM Agent for Adversarial Purification. arXiv:2405.20770 [cs.CL] https://arxiv.org/abs/2405.20770

[20] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large Language Model-Based Agents for Software Engineering: A Survey. *CoRR* abs/2409.02977 (2024). https://doi.org/10.48550/ARXIV.2409.02977 arXiv:2409.02977

[21] Tong Liu, Zizhuang Deng, Guozhu Meng, Yuekang Li, and Kai Chen. 2024. Demystifying RCE Vulnerabilities in LLM-Integrated Apps. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 1716–1730. https://doi.org/10.1145/3658644.3690338

[22] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Zihao Wang, Xiaofeng Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, and Yang Liu. 2024. Prompt Injection attack against LLM-integrated Applications. arXiv:2306.05499 [cs.CR] https://arxiv.org/abs/2306.05499

[23] Bettina Messmer, Vinko Sabolčec, and Martin Jaggi. 2025. Enhancing Multilingual LLM Pretraining with Model-Based Data Selection. arXiv:2502.10361 [cs.CL] https://arxiv.org/abs/2502.10361

[24] Tomas Bueno Momcilovic, Beat Buesser, Giulio Zizzo, Mark Purcell, and Dian Balta. 2024. Towards Assurance of LLM Adversarial Robustness using Ontology-Driven Argumentation. arXiv:2410.07962 [cs.AI] https://arxiv.org/abs/2410.07962

[25] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 97, 13 pages. https://doi.org/10.1145/3597503.3639187

[26] NVD. 2025. National Vulnerability Database. https://nvd.nist.gov/vuln/search. Accessed: 2025-07-09.

[27] Rodrigo Pedro, Miguel E. Coimbra, Daniel Castro, Paulo Carreira, and Nuno Santos. 2025. Prompt-to-SQL Injections in LLM-Integrated Web Applications: Risks and Defenses . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1768–1780. https://doi.org/10.1109/ICSE55347.2025.00007

[28] Raghuveer Peri, Sai Muralidhar Jayanthi, Srikanth Ronanki, Anshu Bhatia, Karel Mundnich, Saket Dingliwal, Nilaksh Das, Zejiang Hou, Goeric Huybrechts, Srikanth Vishnubhotla, Daniel Garcia-Romero, Sundararajan Srinivasan, Kyu J Han, and Katrin Kirchhoff. 2024. SpeechGuard: Exploring the Adversarial Robustness of Multimodal Large Language Models. arXiv:2405.08317 [cs.CL] https://arxiv.org/abs/2405.08317

[29] Mansi Phute, Alec Helbling, Matthew Hull, ShengYun Peng, Sebastian Szyller, Cory Cornelius, and Duen Horng Chau. 2024. LLM Self Defense: By Self Examination, LLMs Know They Are Being Tricked. arXiv:2308.07308 [cs.CL] https://arxiv.org/abs/2308.07308

[30] Security PRIDE. 2025. TaintP2X. https://github.com/security-pride/TaintP2X/blob/main/Taint_Propagation/taint/llm_sources.xlsx. Accessed: 2025-07-09.

[31] Bowen Qin, Binyuan Hui, Lihan Wang, Min Yang, Jinyang Li, Binhua Li, Ruiying Geng, Rongyu Cao, Jian Sun, Luo Si, Fei Huang, and Yongbin Li. 2022. A Survey on Text-to-SQL Parsing: Concepts, Methods, and Future Directions. *CoRR* abs/2208.13629 (2022). https://doi.org/10.48550/ARXIV.2208.13629 arXiv:2208.13629

[32] Libo Qin, Qiguang Chen, Yuhang Zhou, Zhi Chen, Yinghui Li, Lizi Liao, Min Li, Wanxiang Che, and Philip S. Yu. 2024. Multilingual Large Language Model: A Survey of Resources, Taxonomy and Frontiers. arXiv:2404.04925 [cs.CL] https://arxiv.org/abs/2404.04925

[33] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=dHng2O0Jjr

[34] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html

[35] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Yang Liu, and Yingjiu Li. 2025. LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning. arXiv:2401.16185 [cs.CR] https://arxiv.org/abs/2401.16185

[36] Qwen Team. 2024. Function Calling: Qwen Documentation. https://qwen.readthedocs.io/en/latest/framework/function_call.html. Accessed: 2025-07-09.

[37] Simone Tedeschi, Felix Friedrich, Patrick Schramowski, Kristian Kersting, Roberto Navigli, Huu Nguyen, and Bo Li. 2024. ALERT: A Comprehensive Benchmark for Assessing Large Language Models' Safety through Red Teaming. arXiv:2404.08676 [cs.CL] https://arxiv.org/abs/2404.08676

[38] Bertie Vidgen, Nino Scherrer, Hannah Rose Kirk, Rebecca Qian, Anand Kannappan, Scott A. Hale, and Paul Röttger. 2024. SimpleSafetyTests: a Test Suite for Identifying Critical Safety Risks in Large Language Models. arXiv:2311.08370 [cs.CL] https://arxiv.org/abs/2311.08370

[39] Shaw Walters, Sam Gao, Shakker Nerd, Feng Da, Warren Williams, Ting-Chien Meng, Amie Chow, Hunter Han, Frank He, Allen Zhang, Ming Wu, Timothy Shen, Maxwell Hu, and Jerry Yan. 2025. Eliza: A Web3 friendly AI Agent Operating System. *CoRR* abs/2501.06781 (2025). https://doi.org/10.48550/ARXIV.2501.06781 arXiv:2501.06781

[40] Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-Agent-v2: Mobile Device Operation Assistant with Effective Navigation via Multi-Agent Collaboration. 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*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/0520537ba799d375b8ff5523295c337a-Abstract-Conference.html

[41] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024. A survey on large language model based autonomous agents. *Frontiers Comput. Sci.* 18, 6 (2024), 186345. https://doi.org/10.1007/S11704-024-40231-1

[42] Irene Weber. 2024. Large Language Models as Software Components: A Taxonomy for LLM-Integrated Applications. *CoRR* abs/2406.10300 (2024). https://doi.org/10.48550/ARXIV.2406.10300 arXiv:2406.10300

[43] Liangxuan Wu, Chao Wang, Tianming Liu, Yanjie Zhao, and Haoyu Wang. 2025. From Assistants to Adversaries: Exploring the Security Risks of Mobile LLM Agents. *CoRR* abs/2505.12981 (2025). https://doi.org/10.48550/ARXIV.2505.12981 arXiv:2505.12981

[44] Yuhao Wu, Franziska Roesner, Tadayoshi Kohno, Ning Zhang, and Umar Iqbal. 2025. IsolateGPT: An Execution Isolation Architecture for LLM-Based Agentic Systems. arXiv:2403.04960 [cs.CR] https://arxiv.org/abs/2403.04960

[45] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE037 (June 2025), 24 pages. https://doi.org/10.1145/3715754

[46] Zhengmin Yu, Jiutian Zeng, Siyi Chen, Wenhan Xu, Dandan Xu, Xiangyu Liu, Zonghao Ying, Nan Wang, Yuan Zhang, and Min Yang. 2025. CS-Eval: A Comprehensive Large Language Model Benchmark for CyberSecurity. arXiv:2411.16239 [cs.CR] https://arxiv.org/abs/2411.16239

[47] Zhexin Zhang, Leqi Lei, Lindong Wu, Rui Sun, Yongkang Huang, Chong Long, Xiao Liu, Xuanyu Lei, Jie Tang, and Minlie Huang. 2024. SafetyBench: Evaluating the Safety of Large Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 15537–15553. https://doi.org/10.18653/v1/2024.acl-long.830

[48] Yanjie Zhao, Xinyi Hou, Shenao Wang, and Haoyu Wang. 2025. LLM App Store Analysis: A Vision and Roadmap. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 125 (May 2025), 25 pages. https://doi.org/10.1145/3708530

[49] Xin Zhou, Ting Zhang, and David Lo. 2024. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions. arXiv:2401.15468 [cs.SE] https://arxiv.org/abs/2401.15468

[50] Andy Zou, Zifan Wang, J. Z. Kolter, and Matt Fredrikson. 2023. Universal and Transferable Adversarial Attacks on Aligned Language Models. *ArXiv* abs/2307.15043 (2023).