Make Agent Defeat Agent: Automatic Detection of Taint-Style Vulnerabilities in LLM-based Agents

Fengyu Liu[†], *Yuan Zhang*[†], *Jiaqi Luo*[†], *Jiarun Dai*[†], *Tian Chen*[†], *Letian Yuan*[†], *Zhengmin Yu*[†], *Youkun Shi*[†], *Ke Li*[†], *Chengyuan Zhou*[†], *Hao Chen*[‡], *Min Yang*[†]

[†]*Fudan University* [‡]*University of California, Davis*

Abstract

Large Language Models (LLMs) have revolutionized software development, enabling the creation of AI-powered applications known as LLM-based agents. However, recent studies reveal that LLM-based agents are highly susceptible to taint-style vulnerabilities, which allow malicious prompts to exploit security-sensitive operations. These vulnerabilities pose severe threats to the security of agents, potentially allowing attackers to take over the entire agent remotely.

In this paper, we propose a novel directed greybox fuzzing approach, called `AgentFuzz`, the first fuzzing framework for detecting taint-style vulnerabilities in LLM-based agents. `AgentFuzz` consists of three key phases. First, `AgentFuzz` leverages the LLM to generate functionality-specific seed prompts in the form of natural language. Second, `AgentFuzz` utilizes a multifaceted feedback design to assess seed quality from both semantic and distance levels, prioritizing seeds with higher quality. Finally, `AgentFuzz` employs functionality and argument mutator to refine seeds and trigger vulnerabilities effectively. In our evaluation against 20 widely-used open-source agent applications, `AgentFuzz` identified 34 high-risk 0-day vulnerabilities, achieving 33 times higher precision than the state-of-the-art approach. These vulnerabilities encompass serious threats like code injection, impacting 14 open-source agents, with 7 of them having over 10,000 stars on GitHub. To date, 23 CVE IDs have been assigned.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable advancement in various downstream tasks, such as code generation [41,43], question answering [29,39,60], etc. Nowadays, developers are actively integrating LLMs to build AI-powered applications, which are widely known as **LLM-based agents** [65, 68]. These emerging LLM-based agents could understand natural language instructions, perceive external environments, and intelligently carry out various actions.

Currently, the ecosystem of LLM-based agents has rapidly evolved and demonstrates various product forms. A common mode involves deployment on local devices such as desktop software [12], where users can interact directly with the agent to intelligently operate the device. Alternatively, agents may be deployed on centralized remote servers and accessed through websites, allowing users to engage with the agent remotely [24]. These agents typically handle essential tasks, such as executing code and processing sensitive data. For instance, LLM platforms like Coze and GPT host a variety of agents [4, 7], attracting millions of users [78] and storing vast amounts of user privacy data, underscoring the growing importance of their security.

However, sadly, existing studies reveal that these LLM-based agents are vulnerable to serious security threats (e.g., prompt injection [75]), which could potentially cause information leakage, malicious code execution, and so on. Among these, **taint-style vulnerabilities** [49, 54], a well-established concern in traditional code security research [46, 51], are undoubtedly among the most critical types that require attention. These vulnerabilities stem from developers' over-reliance on LLM outputs and failure to sanitize harmful content before passing it to security-sensitive operations (SSO). This oversight allows malicious payloads embedded in prompts to flow into SSOs, triggering vulnerabilities like code injection and allowing attackers to achieve local privilege escalation or even gain remote control of the agent. While recent studies [49] have shed light on detecting taint-style vulnerabilities in agents, their approaches such as using static analysis to identify source-to-sink call chains still suffer from high false-positive and false-negative rates.

Therefore, in this work, we are highly motivated to design a vulnerability detection approach that can effectively vet the security of real-world popular LLM-based agents against taint-style vulnerabilities. Considering the fact that taint-style vulnerabilities can only be triggered at specific sinks, directed fuzzing has long been embraced for its ability to target testing efforts towards given code locations, thereby increasing the likelihood of discovering taint-style vulnerabilities. Hence, it should be an appealing solution to migrate conventional directed fuzzing techniques to LLM-based agents. However,

it is definitely a non-trivial task, due to the complex nature of LLM-based agents (detailed as follows):

- *C1: How to generate functionality-specific seeds in the form of natural language?* Unlike traditional applications with structured inputs, LLM-based agents receive natural language prompts, which the LLM interprets to invoke specific functionalities. Traditional fuzzing techniques are designed for structured data and struggle to generate natural language seed prompts for the diverse functionalities of agents.
- *C2: How to prioritize high-quality seeds during the fuzzing procedure to boost the fuzzing efficiency?* Prioritizing high-quality seeds more likely to trigger the sink can improve fuzzing efficiency. Traditional fuzzing techniques use metrics like CFG distances to prioritize seeds, assuming seeds closer to the sink are more promising. However, this heuristic faces challenges in agents due to flexible code features, and distance alone fails to capture the semantic gap between seeds, making it less reliable for evaluating seed quality.
- *C3: How to effectively mutate seeds to trigger the taint-style vulnerability?* Once high-quality seeds are selected, the next step is to mutate them to trigger the vulnerability. To achieve this, the seed must possess specific natural language semantics to invoke functionality containing the sink. Traditional mutators, however, are designed for byte-level mutations and fail to adjust the semantics of the seed. Moreover, various constraints are typically present along the path to the sink in the agent's code. Identifying and mutating the specific parts of the prompt that can satisfy these constraints presents a significant challenge.

In this paper, we propose a novel directed greybox fuzzing approach, called `AgentFuzz`, the first fuzzing framework for detecting taint-style vulnerabilities in LLM-based agents. Our approach is motivated by several key insights that help address the proposed challenges. First, agents use various components (e.g., tools) to process actions. The class and method names of these components encompass rich natural language semantics that reflect their functional purpose, helping generate functionality-specific seed prompts. Second, high-quality seeds share semantics aligned with the functionality of the vulnerable component and meet the necessary code constraints, enabling the agent to invoke specific vulnerable components and trigger the vulnerability. Third, the semantics embedded in the method and class names of vulnerable components serve as guidance for functionality mutations. Additionally, the overlap between the user prompt and the runtime argument of the vulnerable component helps identify the specific section of the prompt that needs mutation to meet constraints, enabling deterministic mutations.

Based on these insights, we design `AgentFuzz` with three main phases. In the first phase, `AgentFuzz` extracts call chains leading to predefined sinks such as SQL injection and code injection. It then employs an LLM-assisted approach to interpret the natural language semantics embedded in the method and class names within these chains, thereby generating functionality-specific seed prompts. In the second phase, `AgentFuzz` evaluates seed quality through a multifaceted feedback strategy, prioritizing seed based on its execution traces, semantic consistency, and CFG distances to the sink. This approach helps efficiently focus on the most promising seed for triggering vulnerabilities. In the third phase, `AgentFuzz` employs functionality and argument mutators to refine the seed. It autonomously selects the most suitable mutator based on runtime feedback, thus generating semantically-correct and constraint-valid seed prompts. After the mutation, `AgentFuzz` uses predefined oracles to determine whether the mutated prompts successfully trigger the vulnerabilities.

To demonstrate the effectiveness and performance of `AgentFuzz`, we evaluated it on 20 widely used open-source agent applications that provide web services, which are more prone to attacks due to their exposure and potential impact when exploited. Our evaluation shows that `AgentFuzz` successfully identified 34 0-day taint-style vulnerabilities (24 of which were undetected by existing approaches), achieving a precision rate of 100%, outperforming the state-of-the-art approach LLMSmith [49] by 33 times. These vulnerabilities include high-risk issues such as code injection and SQL injection, affecting 14 open-source agents, 7 of which have over 10k stars on GitHub. All 34 vulnerabilities were confirmed to be exploitable. The discovered vulnerabilities pose severe security risks, enabling attackers to take full control of the servers and potentially causing significant financial losses. Considering the substantial security impact, we responsibly reported them to the developers of the affected agents. As of now, 23 CVE IDs have been assigned.

In summary, the paper makes the following main contributions:

- We propose the first directed fuzzing approach to detect taint-style vulnerabilities for LLM-based agents.
- We have developed a prototype of this approach, named `AgentFuzz`, which effectively detects taint-style vulnerabilities in real-world agent applications.
- We evaluated `AgentFuzz` on 20 real-world popular agent applications. `AgentFuzz` successfully identified 34 0-day vulnerabilities. As of now, these vulnerabilities have been assigned 23 CVE IDs.

2 Problem Statement

In this section, we present an overview of the LLM-based agents (in §2.1) and define the taint-style vulnerabilities within them (in §2.2).

2.1 LLM-based Agents

With the rapid advancement of LLMs, LLM-based agents have undergone significant development. By leveraging the

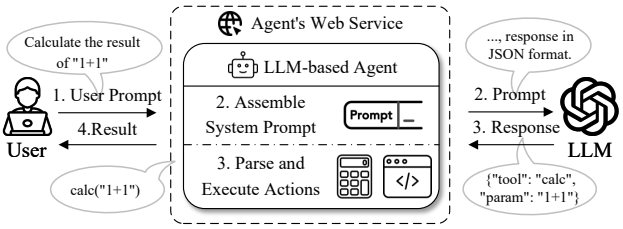

Figure 1: Simplified workflow of LLM-based agents.

powerful generative abilities of LLMs and various components such as externally provided tools, agents can comprehend end users' natural language instructions (i.e., user prompts) and handle more complex tasks beyond the capabilities of a standalone LLM [31]. In practice, LLM-based agents are frequently employed to perform critical tasks like executing code, handling sensitive data, and automating business workflows [65], all of which involve invoking various sensitive functions. A notable example is DB-GPT [70], which handles data processing tasks and accesses substantial amounts of user data, enabling it to achieve complex data analysis.

Figure 1 illustrates the workflow of LLM-based agents. Specifically, (1) end users interact with the agent through user prompts, e.g., "calculate the result of 1+1." (2) The agent processes the user prompt and assembles it with system prompts defined by the developer. The system prompt typically specifies the return format for the LLM, enabling the agent to effectively interpret the LLM's response. The assembled prompt is then sent to the LLM to interpret the user's intent and plan the corresponding actions. (3) The agent leverages the output parser component to analyze the response according to the agreed-upon format and extract the action planned by the LLM. It then adheres to these instructions to invoke specific components and execute the intended action, e.g., invoking the `calc` tool to compute `1+1`. (4) Finally, the agent returns the result of the action to the user.

2.2 Taint-style Vulnerability in Agents

Taint-style vulnerabilities are a widespread type of flaw in traditional applications and consistently rank high on the OWASP Top Ten list [14, 15]. These vulnerabilities typically occur when malicious user input is passed into security-sensitive operations without proper validation, potentially leading to critical issues such as code and SQL injection.

2.2.1 Real-world Example

We use a real-world code injection vulnerability (i.e., CVE-2024-5**93, anonymized for ethical reasons) that we discovered in a popular open-source agent to demonstrate the proposed taint-style vulnerability in agents. Figure 2 presents a simplified code snippet illustrating the vulnerability. To exploit this vulnerability, the attacker sends a malicious prompt

Attack Prompt Payload

Use **Elasticsearch** for a **similarity search** with **permission checks** to find documents with **'source_doc:**print(1)**'.**

```
1  Tools = [ ElasticSearch(), WebSearch(),
                    ElasticsearchPermissionCheck() ]
2  @router.post('/chat')
3  def assistant_agent(prompt):                    # User input        ①
4      resp = llm.invoke(OpenAI(), prompt)         # LLM response      ②
       # tool = ElasticsearchWithPermissionCheck()
5      tool = Tools.get(resp["tool"])              # Get tool
       # resp["content"] = "source_doc:print(1)"
6      result = tool.run(resp["content"])          # Invoke tool with
                                                   # indirect call     ③
7  Class ElasticsearchPermissionCheck():
8      def similarity_search(self, content):
           # content = "source_doc:print(1)"
9          if "source_doc" in content:
10             return eval(content.split(':')[1])  # eval('print(1)') ④
```

Figure 2: A real-world vulnerability example from a popular open-source agent (the vulnerability has been patched).

to the agent via the web service in lines 2-3 (i.e., source), a portion of which is returned by the LLM and directly passed to the `eval` function in line 10 (i.e., sink) without adequate sanitization, resulting in code injection.

Technically, the whole vulnerability exploitation process involves four steps. ❶ The attacker sends an HTTP request with a crafted prompt to the agent via its web API (lines 2-3), e.g., `http://agent/chat?prompt=${prompt}`. ❷ The agent forwards the attack prompt to the LLM, which plans and generates the appropriate actions for the agent to execute (line 4). Given that the intent of the attack prompt is to search a document with permission checks, the LLM selects the `ElasticsearchPermissionCheck` tool from the three tools defined in line 1 and sends it back to the agent. ❸ The agent processes the LLM's response and executes the specified action. As shown in line 6, the agent invokes the `ElasticsearchPermissionCheck` tool via an indirect call (i.e., `run` method). ❹ The `ElasticsearchPermissionCheck` tool executes with "`source_doc:print(1)`" as its argument, which actually is a specific string in the user prompt and parsed from the LLM response. As illustrated in lines 9-10, since the argument meets the if-condition, it is processed by the `split` method and flows into the sink, i.e., `eval("print(1)")`, triggering a code injection vulnerability.

2.2.2 Threat Model

In our threat model, we assume agents and their developers are benign and assume their running environments are not compromised. We consider attackers to be malicious users

who possess the capability to interact with an agent under normal operational conditions. By sending crafted prompts to the agent, an attacker could control the execution of a sensitive function (i.e., sink) and gain unauthorized privileges. The threat model covers two kinds of attackers: *remote* attackers and *local* attackers. Remote attackers interact with the agent through its web service, using web APIs to input malicious prompts and gain control of the server by executing harmful code. Such web services are common in LLM-based agents. For example, popular agents like AutoGPT [2] and Dify [6] provide users with web APIs to input prompts. Local attackers operate on the same device as the agent, gaining local access to interact with high-privilege agents through malicious prompts. These prompts can trigger sensitive functions in locally deployed agents (e.g., MobileAgent [64]), leading to privilege escalation and other malicious activities.

Note that though an agent may run in an isolated environment (e.g., deployed in a Docker container), taint-style vulnerabilities still pose significant security risks. On one hand, the attacker could leverage a code execution vulnerability to steal sensitive data within the agent's running environment (e.g., the LLM API key used by the agent may be leaked to allow unauthorized requests to the LLM, causing financial losses to the maintainers of the agent). On the other hand, the attacker could use these vulnerabilities to further compromise the agent's integrity and availability. For example, the attacker could access the database and delete critical user data, or launch a DoS attack to disrupt the agent's functionality.

3 Related Work

In this section, we review related works in taint-style vulnerability detection and summarize their limitations to motivate our research.

Vulnerability Detection of LLM-based Agents. Taint-style vulnerabilities pose a significant threat to the security of agents, highlighting the critical need for effective detection and mitigation strategies. However, to the best of our knowledge, no existing research has introduced a fully automated and accurate approach to detect taint-style vulnerabilities in LLM-based agents. The most relevant work, LLMSmith [49], applies static analysis techniques to identify call chains that start from user-level APIs and terminate at sink callsites, reporting these chains as vulnerabilities. Nevertheless, its coarse-grained call graph analysis, along with the prevalence of indirect calls in Python, leads to high false positive and false negative rates. Other studies mainly emphasized threats such as prompt leaking [45,47,55] and jailbreaking [56,71,72,79], which represent entirely different vulnerability patterns.

Taint-Style Vulnerability Detection. In fact, taint-style vulnerability detection has been studied extensively for decades [46]. Existing works can be broadly categorized into static and dynamic approaches.

Existing static analysis approaches [30, 46, 48, 51] treat variables representing user inputs as sources and parameters of security-sensitive operations as sinks. They then perform dataflow analysis to detect potential vulnerabilities by identifying source-to-sink paths. However, these methods have significant limitations. Firstly, they cannot generate proofs of concept (PoC) to verify exploitability, leading to a high rate of false positives. Secondly, agents are typically developed in Python, a dynamically typed language that allows developers to use indirect calls and other advanced techniques to implement complex functionalities more easily. While these features enhance flexibility, they complicate static analysis, increasing false positives and missed detections.

Existing dynamic approaches [1,37,40,58,61], on the other hand, typically employ fuzzing techniques to detect vulnerabilities (i.e., fuzzers). Specifically, fuzzers generate initial seeds that conform to the target application's input format and mutate them using mutation strategies such as bitflip [1]. They then leverage evidence within the code (e.g., distance to sink [32]) to narrow the exploration space and generate seeds that reach the target sink, triggering the vulnerability. However, unlike traditional applications with structured inputs, agent inputs are prompts that lack a fixed structure and encompass the entire spectrum of natural language. These prompts carry rich natural language semantics, empowering the agent to invoke various functionalities intelligently. Obviously, traditional fuzzers like AFL [1] cannot generate inputs with specific semantics or mutate their meaning, thereby failing to explore the diverse functionalities within agents.

4 Overall Idea

The rapid development of agents and the gaps in existing works within this field further highlight the urgent need for an effective approach to detect taint-style vulnerabilities in agents. Fuzzing techniques, with their high precision and ability to generate PoCs for vulnerability verification, have greatly attracted our interest. Moreover, taint-style vulnerabilities are characterized by a fixed sink in the vulnerable code, meaning their detection is fundamentally a directed fuzzing task.

Therefore, in this paper, we utilize the Directed Greybox Fuzzing (DGF) [32] technique to address the problem of taint-style vulnerability detection in agents. In this section, we illustrate the challenges encountered and present our key insights (in §4.1). Then, we introduce our approach overview for detecting vulnerability within agents (in §4.2).

4.1 Challenges & Insights

Traditional DGF techniques [1, 32, 73] typically comprise the following three modules. The seed generation module creates and adds valid seeds to the seed pool. The scheduling module selects a seed from the seed pool for mutation, after which the mutation module modifies it and updates the seed pool, thereby iterating through the entire fuzzing loop.

Given the success of existing fuzzing techniques, adapting these well-established approaches to LLM-based agents seems an appealing idea. However, we argue that directly applying them to agents presents significant challenges.

Challenge I: How to generate functionality-specific seeds in the form of natural language? Traditional fuzzing techniques primarily target structured inputs and rely on public seed datasets [32, 33], which clearly cannot generate the natural language prompts required by agents. While LLM-driven methods have shown promise for seed generation [52, 69], existing works fail to produce functionality-specific prompts needed to adequately explore the diverse functionalities of agents (e.g., chat, code generation), thus limiting their effectiveness in fuzzing agent applications.

Solution: To address this, we propose an LLM-assisted seed generation approach. The key insight is that, as outlined in Langchain [9], agents utilize various components (e.g., vector storage and tools) to process actions based on the prompt's intent. Developers typically convey the functional purpose of components through their class and method names, allowing the LLM to interpret their functionality and, based on the user prompt's intent, invoke the appropriate component. As shown in Figure 2, the `ElasticsearchPermissionCheck` component's name conveys clear natural language semantics that describe its functionality. Therefore, we leverage LLM's advanced natural language understanding capabilities to interpret the semantics embedded in components, thereby generating functionality-specific seed prompts.

Challenge II: How to prioritize high-quality seeds during the fuzzing procedure to boost the fuzzing efficiency? Seed scheduling strategies [1, 32, 73] prioritize high-quality seeds from a vast number of candidates, significantly enhancing fuzzing efficiency. Common directed fuzzing approaches suggest that CFG distances could serve as valuable feedback for seed scheduling [32]. That is, seeds whose execution traces have shorter distances towards the sink are more likely to trigger the vulnerability, making them higher quality. At first glance, distance-based scheduling seems applicable to agent fuzzing, as taint-style vulnerabilities are triggered at specific sinks, and seeds closer to these sinks should be prioritized. However, we argue that it is not directly applicable to LLM agents. The key reasons are two-fold.

❶ *Indirect calls.* Indirect calls are pervasive in agents, making it difficult to construct a complete call graph, let alone a control flow graph. Consequently, static analysis techniques cannot reliably assess the distance to the sink, making distance-based seed quality evaluation unfeasible. ❷ *Semantic proximity.* Even if the distance to the sink is accurately determined, relying solely on distance still fails to reflect the true quality of the seed. As shown in Figure 2, although the CFG distance to the sink may be the same for both `ElasticSearch` and `WebSearch`, `ElasticSearch` is semantically closer to `ElasticsearchPermissionCheck`, which contains the sink. Thus, seeds invoking the `ElasticSearch` are of higher quality than those invoking `WebSearch`, highlighting the inadequacy of distance-based evaluation in agent contexts.

Solution: Therefore, to evaluate seed quality in the context of agents, we propose a multifaceted feedback strategy that assesses seed quality from both semantic and distance levels. At the semantic level, a high-quality seed must align with the functionality of the vulnerable component in terms of natural language semantics, thereby ensuring LLM can interpret the prompt's intent and invoke the target vulnerable component. At the distance level, the execution trace of high-quality seeds should reach blocks closer to the sink callsite. As illustrated in Figure 2, the attack prompt (1) invokes the vulnerable component through its specific semantics, and (2) reaches blocks closer to the sink. Unlike existing approaches [49] that rely on statically constructing complete call chains, which may fail due to indirect calls. Our approach leverages the call chain semantics to schedule prompts that dynamically reach the sink. This direct fuzzing strategy mitigates indirect call issues and increases the chance of triggering the sink.

Challenge III: How to effectively mutate seeds to trigger the taint-style vulnerability? With high-quality seeds in hand, triggering vulnerabilities requires the seed prompt to (1) include specific semantics that invoke the vulnerable component (e.g., `ElasticsearchPermissionCheck`) among various agent components, and (2) satisfy multiple constraints for the component's arguments (e.g., ensuring the `content` argument includes the string `source_doc:`).

However, traditional mutators [1, 32, 73] are unable to address these requirements. On the one hand, they primarily focus on random byte-level transformations (e.g., bit flips [1]) rather than mutating the natural language semantics of seed prompts, thus failing to generate prompts with the specific semantics required to invoke vulnerable components. On the other hand, while traditional mutators can solve constraints, they cannot identify which specific section of the prompt needs mutation to enable the component's argument, derived from the LLM's response, to satisfy constraints.

Solution: To address this challenge, we designed two mutators. ❶ The functionality mutator refines the seed's natural language semantics using the call chain of the vulnerable component, helping mitigate indirect call issues and generate prompts that effectively invoke components with the desired functionality. ❷ The argument mutator solves constraints and matches the argument's value to the user prompt through runtime substring matching, thereby determining which section of the prompt requires alteration to satisfy these constraints. This design is based on our observation that, to perform a specific task, a user prompt typically comprises two key elements: the action and associated data to be processed. The LLM interprets the action from the prompt and provides the associated data as arguments to the component, thereby completing the task. By modifying specific sections of the user prompt, we can control the arguments passed to the component and adjust the value of related variables.

Algorithm 1: Fuzzing Loop

Input: Agent Code, C
Output: Vulnerability PoCs, P

```
1  S ← ExtractSinkCallsites(C)
2  for sink_callsite in S do
3      seed_pool ← ∅
4      call_chains ← StaticAnalyze(sink_callsite)
5      for call_chain in call_chains do
6          initial_seed ← GenerateSeed(call_chain)
7          feedback ← Execute(initial_seed)
8          seed_pool ← seed_pool ∪ {seed, feedback}
9      while not Timeout do
10         seed, feedback ← SelectSeed(seed_pool)
11         mutator ← SelectMutator(seed, feedback)
12         new_seed ← Mutate(mutator, seed)
13         feedback ← Execute(new_seed)
14         seed_pool ← seed_pool ∪ {new_seed, feedback}
15         if VulnOracle(new_seed) == Success then
16             P ← P ∪ {new_seed}
17             break
```

4.2 Approach Overview

Drawing on the above ideas, we introduce a novel directed greybox fuzzing approach to effectively detect taint-style vulnerabilities within agents. As outlined in Algorithm 1, the overall fuzzing loop is primarily divided into three phases: seed generation, seed scheduling, and seed mutation.

Specifically, in the first phase, our approach identifies sink callsites within the agent (line 1), with each callsite representing a separate fuzzing loop (line 2). Our approach then extracts all call chains leading to the sinks and inputs them into LLM, using the CoT [66] reasoning strategy to generate functionality-specific seeds (lines 4–6). In the second phase, our approach inputs the generated seed prompts into the agent for execution and adds them to the seed pool (lines 7-8). Then, our approach scores each seed based on three factors (semantic score, distance score, and penalty score) and prioritizes the seed with the highest score for further mutation (line 10). In the third phase, our approach uses two mutators to mutate selected seeds (line 12). For mutator scheduling, our approach leverages the LLM to autonomously select the appropriate mutator (line 11). After mutation, our approach executes the mutated seeds in the agent and uses predefined vulnerability oracles to determine if they trigger the sink, outputting successful prompts as vulnerability PoCs (lines 13-17).

5 AgentFuzz Design

In this section, we provide the design details of our approach, called AgentFuzz. As shown in Figure 3, AgentFuzz primarily consists of three phases.

- *LLM-assisted Seed Generation (§5.1)* generates seed prompts with natural language semantics.
- *Feedback-driven Seed Scheduling (§5.2)* evaluates seed quality based on the multifaceted feedback.
- *Sink-guided Seed Mutation (§5.3)* uses two mutators to mutate and generate higher-quality seed.

5.1 LLM-assisted Seed Generation

In this phase, AgentFuzz uses static analysis techniques to extract call chains of each sink callsite, and employs the LLM to generate seed prompts with natural language semantics.

5.1.1 Sink Callchain Extraction

As outlined in our insights (§4.1), the class and method names of a component provide crucial natural language semantics that enable the LLM to infer its functionality. Consequently, AgentFuzz identifies sink callsites within the agent and traces the call chain backward, following the callers and extracting class and method names along the way.

Specifically, we manually construct a predefined set of sinks, each sink defined by a unique method signature, i.e., <*package, class, method, parameters*>. These predefined sinks, as listed in Table 5 of Appendix C, encompass common types of security-sensitive operations recognized in the OWASP Top Ten [14, 15], such as code injection, command injection, and SSRF. After that, AgentFuzz employs a depth-first approach to traverse the call graph, iteratively tracing backward from the sink callsites to identify the callers, thereby extracting all call chains that can reach a specific sink callsite. For each call chain, AgentFuzz terminates the backward tracing once no further callers can be identified in the constructed call graph. Take Figure 2 as an example, AgentFuzz starts the backward tracing from the `eval` function, and the final extracted call chain is `eval→ ElasticsearchPermissionCheck.similarity_search`.

5.1.2 LLM-assisted Seed Prompt Generation

Next, AgentFuzz leverages the one-shot learning [62] technique with Chain of Thought (CoT [66]) reasoning strategy to guide the LLM in generating seed prompts based on the semantics of the extracted call chain. A given callsite may have multiple call chains, so we generate a distinct seed prompt for each one. The prompt example is presented in Figure 4.

- **One-shot learning** enables the LLM to imitate reasoning logic from an example to perform similar tasks [62]. In AgentFuzz, the example consists of the call chain to the sink, a sample prompt that triggers the sink, and the reasoning process behind generating this sample prompt based on the call chain. This enables LLM to understand and imitate the process of inferring natural language semantics from the call chain and generating functionality-specific prompts.

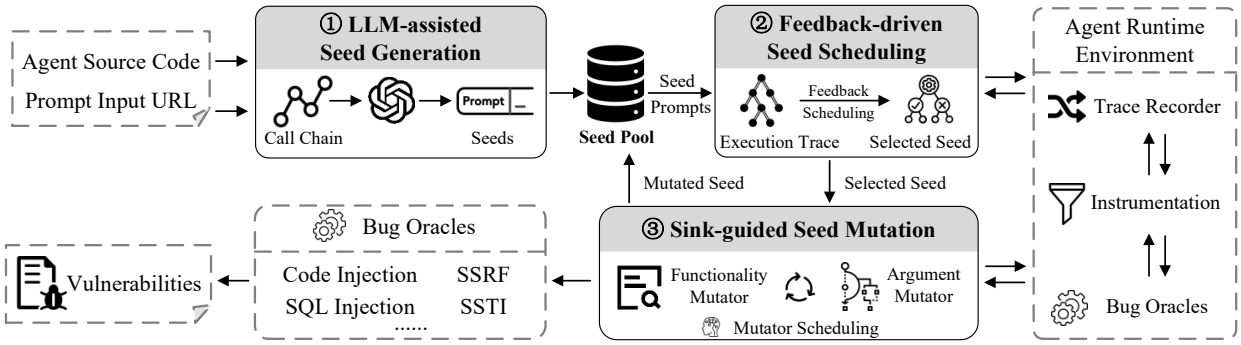

Figure 3: Architecture of AgentFuzz.

- **CoT** guides the LLM in refining its logic through step-by-step reasoning [66]. This part involves: (1) inferring the component's functionality from the call chain, (2) generating a prompt aligned with the inferred functionality, and (3) performing self-verification [67] to ensure prompt intent matches the functionality and modifying it if necessary.

Seed Generation Prompt

You need to craft a prompt that guides the agent to invoke this component. **First**, infer the component's functionality from the class and function names in the call chain. **Second**, generate a prompt with matching semantics. **Finally**, verify its semantic similarity to the component, and revise it if necessary.

Now here is an example you can follow:
INPUT: <CALL CHAIN>: calculator→eval
OUTPUT: <COT>: (1) The call chain "calculator→eval" suggests that the component is designed to evaluate mathematical expressions. (2) The prompt should instruct the LLM to use the calculator component to perform an evaluation that aligns with the function inferred from the call chain. (3) The prompt should specify an expression for evaluation to ensure it is directly related to the component's purpose. <PROMPT>: Please use the calculator to evaluate the following expression: 3 * (4 + 5).

Figure 4: Seed generation prompt.

5.2 Feedback-driven Seed Scheduling

In this phase, we first introduce our novel multifaceted feedback design and then explain how AgentFuzz utilizes the feedback to prioritize the seed with higher quality.

5.2.1 Multifaceted Feedback Design

AgentFuzz measures the quality of the seed based on the following factors: ❶ semantic score, i.e., using an LLM to assess the semantic consistency between the prompt and the vulnerable component, ❷ distance score, i.e., employing the distance in CFG to evaluate whether the execution trace of the seed prompt is closer to the sink. Moreover, to avoid local convergence, AgentFuzz also considers ❸ penalty score, i.e., using the number of times the seed and call chain have been selected to assess whether the seed has fallen into a local optimum. Finally, the feedback score of the seed is as follows:

$$F_s = \alpha S_s + \beta D_s - P_s \tag{1}$$

S_s, D_s, and P_s represent the semantic score, distance score, and penalty score, respectively. α and β are hyper-parameters that can adjust the weights of these factors. A seed with a higher F_s score indicates higher quality.

Semantic Score. Evaluating the natural language semantic quality of a seed prompt solely based on the prompt itself is difficult due to the limited context it provides. To address this, AgentFuzz records the execution trace when the agent executes the seed prompt. It then uses the LLM to analyze the natural language semantic gap between the name of the methods in the execution trace and the call chain leading to the vulnerable component, allowing for a more accurate evaluation of the prompt's semantic score. Our prompts are provided in Figure 5.

Scoring Prompt

You are tasked with evaluating whether the prompt semantics can correctly trigger the target component. Follow these steps: **First**, infer the component's functionality from the given call chain. **Second**, analyze the execution trace's semantics to assess how well it aligns with the target component, and score the prompt.
Scoring Criteria:
10 (Fully Aligned): The prompt perfectly triggers the target.
8-9 (High Semantic Match): The prompt does not directly trigger any component in the call chain, but is semantically close.
1-3 (Low Semantic Match): ...
0 (Completely Unrelated): The semantics of the triggered execution trace are completely unrelated to the target component.

Figure 5: Scoring prompt.

Specifically, for a given seed, AgentFuzz first provides the LLM with its corresponding call chain, helping the LLM to better understand the vulnerable component's functionality. Next, AgentFuzz records the execution trace of the prompt and provides the LLM with key elements such as method and

class names. These elements, rich in natural language semantics, help the LLM fully understand the execution results of the prompt, enabling it to assess the prompt's semantic quality. Finally, AgentFuzz leverages the LLM to evaluate the semantic score of the prompt, which is based on how closely the natural language semantics of the execution trace align with those of the call chain. The greater the alignment, the higher the semantic score, which ranges from 0 (completely unrelated) to 10 (fully aligned). To ensure more consistent scoring, the LLM's temperature is set to 0, which helps produce more uniform responses during evaluation.

Distance Score. The distance score measures how closely the execution trace of the seed prompt is to the sink callsite, serving as a key indicator of seed quality. To evaluate this, AgentFuzz calculates the shortest control flow path to the sink callsite in the CFG for all method calls within the execution trace, taking the smallest of these as the overall distance from the execution trace to the sink. The smaller the distance to the sink, the higher the quality of the seed. Note that a method call absent from the CFG may imply the presence of indirect calls that depend on specific semantics for execution, and as such, AgentFuzz treats its distance as infinite. Finally, the distance score is calculated as:

$$D_s(x) = x^{-k} \tag{2}$$

where k is the hyper-parameter that can adjust the weight and x is the shortest distance of the execution trace to the sink callsite. We calculate the distance score using an inversely proportional function because a shorter distance indicates that the prompt satisfies the required natural language semantics and successfully triggers the vulnerable component. Conversely, as the distance increases, the semantic alignment weakens, making the prompt less likely to invoke the target component. Thus, as x approaches infinity, D_s asymptotically tends to 0.

Penalty Score. Selecting the same seed multiple times could lead to local convergence, reducing fuzzing efficiency. To mitigate this, we decrease the seed's score each time it or its call chain is selected. The penalty score is calculated as:

$$P_s = \gamma S_f + \eta C_f \tag{3}$$

where S_f and C_f denote the selection counts of the seed and its corresponding call chain, respectively. γ and η are hyper-parameters that adjust the weights of these factors.

5.2.2 Feedback-driven Seed Scheduling

Then, leveraging the multifaceted feedback scoring strategy, AgentFuzz selects high-quality seeds from the seed pool for mutation. We visualize this scheduling process in Figure 6. For the call chain ElasticsearchPermissionCheck .similarity_search→eval, there are four corresponding seeds in the seed pool. For S_3, its prompt contains the semantic of search, permission, and check, enabling the LLM

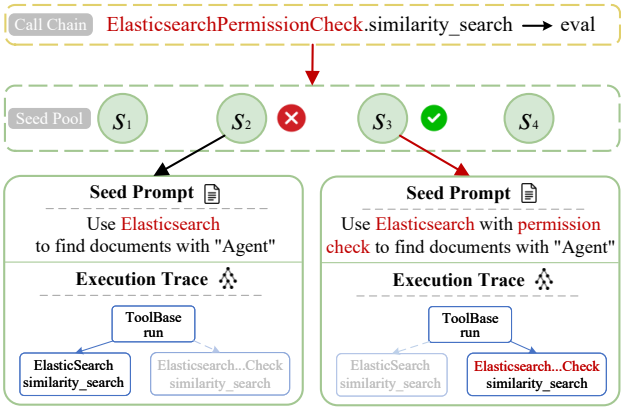

Figure 6: An illustration of the seed scheduling process.

to accurately interpret its intent and invoke the target component, ElasticsearchPermissionCheck. As a result, the execution trace of S_3 closely aligns with the call chain leading to the vulnerable component in both semantic and distance metrics, leading AgentFuzz to assign it a higher score and select for further mutation. In contrast, S_2 lacks the necessary natural language semantics, preventing it from invoking the vulnerable component and resulting in a less consistent execution trace. AgentFuzz thus does not select it.

5.3 Sink-guided Seed Mutation

In this phase, we first introduce our functionality and argument mutators, and then describe how AgentFuzz schedules these mutators to generate higher-quality seeds.

5.3.1 Functionality Mutator

Seed prompts may fail to trigger the vulnerable component due to the semantic gap (e.g., the seed S_2 shown in Figure 6 lacks the semantic of PermissionCheck, causing the agent to invoke the incorrect component). Therefore, we employ a self-improvement [44] mechanism to iteratively mutate and improve the prompts, narrowing the semantic gap between the prompt and the functionality of the vulnerable component.

Self-Improvement Mechanism. Leveraging the LLM's contextual understanding capabilities, AgentFuzz improves the seed based on successful mutations from previous iterations while avoiding errors from earlier iterations. Specifically, AgentFuzz maintains a separate chat session for each seed, where it stores the memory of all mutation attempts associated with that seed. This memory includes details such as the newly generated prompt, its reasoning process, the execution trace, and the corresponding feedback score from previous iterations. Drawing on the stored context, when a seed requires mutation, AgentFuzz interacts with the LLM through the corresponding chat session, iteratively refining the prompt and updating the memory to guide further mutations. Prompts

for functionality mutator are provided in Figure 7.

Functionality Mutator Prompt

You are good at modifying prompts so that their semantics can better help agents understand and invoke specific components.
First, you need to evaluate the intent of the seed prompt based on the execution trace and infer the functionality of the target component based on the call chain.
Second, you need to determine which part of the input prompt causes the agent to call the incorrect component.
Third, you need to base your reasoning on the generated prompt in the history. Don't make the same mistake again.
Finally, based on the previous discovery and your previous reason, you need to modify the prompt to make the prompt semantics more similar to the target call chain and ensure that the agent calls the current component.

Figure 7: Functionality mutator prompt.

5.3.2 Argument Mutator

As discussed in §4.1, beyond ensuring natural language semantic consistency, the seed prompt must ensure that the component's argument, derived from the LLM's response, meets specific constraints, ultimately reaching the sink. For example, as shown in Figure 2, the argument `content` must contain the string `source_doc:` to trigger the vulnerability. To this end, `AgentFuzz` adopts a concolic execution-based approach to solve constraints within the agent's code.

Specifically, first, `AgentFuzz` uses static analysis to extract the expected control-flow path to the sink. By comparing it with the actual execution trace, `AgentFuzz` locates the first unsatisfied conditional statement. Second, `AgentFuzz` initiates concolic execution from the enclosing component of the unsatisfied condition, treating each argument as a symbolic variable. During the execution, the bytecode is interpreted in a custom symbolic environment [34], which runs with concrete values but collects the symbolic expressions of each variable. When the execution reaches the unsatisfied conditional statement or the final sink function, `AgentFuzz` generates the constraints that the condition variable or the argument of the sink function should meet based on their symbolic expressions. Third, `AgentFuzz` applies *Constraint Solving* to generate satisfying inputs for the component's symbolic arguments. Finally, `AgentFuzz` applies *Prompt-to-Argument Mapping* to map and replace the solved values back into the original prompt, thus generating a new prompt for testing. In the following, we clarify the details of *Constraint Solving* and *Prompt-to-Argument Mapping*.

Constraint Solving. To solve the collected constraints, `AgentFuzz` uses the Z3 solver [27] to compute a satisfying value for the symbolic variables (i.e., arguments of the component). In this process, `AgentFuzz` first keeps one variable symbolic but substitutes the others with their concrete values; if solving fails, it gradually increases the number of symbolic variables until a solution is found. Besides, since string operations commonly appear in constraints, `AgentFuzz` models standard Python string operations [17] (e.g., `split`, `startswith`) as part of the constraint-solving process.

We use two examples in Figure 2 (line 9 and line 10) to illustrate the solving process. For the condition "`source_doc`" `in content`, `AgentFuzz` collects the constraint that `content` must contain "`source_doc`", and solves it by generating a value for `content` that includes the required keyword. For the sink argument `content.split(':')[1]`, `AgentFuzz` extracts the constraint that the second element after splitting `content` by `':'` must match a specific value, and solves it by generating a string like "`:value`" to satisfy the constraint.

Prompt-to-Argument Mapping. `AgentFuzz` then adjusts the original prompt based on the value of the newly solved component argument to satisfy the collected constraints. A key challenge here is determining which part of the prompt needs to be modified to meet the constraints. Based on our insights, user prompts for specific tasks generally consist of two main parts: the action and the data to be processed. The LLM interprets the prompt to invoke the component for performing the action and utilizes the provided data in the prompt as arguments for the component. As a result, we can modify the data part of the prompt to control the component's arguments and the variable values in the associated constraints.

To achieve this, `AgentFuzz` employs the Longest Common Substring Matching (LCSM [10]) algorithm to map the variable values in constraints to the specific part of the prompt, then replacing the solution back into the prompt. Specifically, `AgentFuzz` extracts the value of variables from the unsatisfied constraints and uses the LCSM to identify the part of the prompt that determines these variable values. Then, `AgentFuzz` replaces this part with the solved value that meets the constraint. As shown in Figure 2, `AgentFuzz` inserts `source_doc:` into the prompt based on the constraint solution for the `content` argument, thereby triggering the sink.

5.3.3 Mutator Scheduling

`AgentFuzz` employs the LLM to schedule two mutators and autonomously select the appropriate mutator based on the execution context of the seed prompt. Specifically, `AgentFuzz` provides the LLM with the following factors. (1) `AgentFuzz` compares the execution trace with the control flow path to the sink and identifies the conditional statement closest to the sink that is shared by both paths. It then provides the entire condition of this statement to the LLM. (2) `AgentFuzz` extracts the runtime values of variables used in these conditions and provides them to the LLM, helping to determine whether the seed prompt encounters unsolved variable constraints. (3) `AgentFuzz` provides a detailed feedback score to the LLM, aiding in the evaluation of potential semantic inconsistencies with the target component. Note that the CFG is typically incomplete due to indirect calls, resulting in cases where the

execution trace does not overlap with the control flow to the sink callsite. This indicates the natural language semantics of the prompt are insufficient to guide the LLM to invoke the target component through indirect calls. Therefore, in these cases, AgentFuzz prioritizes the functionality mutator to bridge the semantic gap rather than attempting to solve constraints. Mutator scheduling prompt shown in Figure 8.

Mutator Scheduling Prompt

You are a scheduling system tasked with determining which mutator is required to mutate the seed so that it triggers the sink. We provide you with the following information:
Conditional Statement: The last conditional statement encountered and its current state during the agent's execution.
Variable: The variable names and values within the conditions.
Feedback: The previous evaluation result of the seed, including the semantic score and distance score.

Here is the information about the mutators:
Functionality Mutator: Designed to narrow the natural language semantic gap between the prompt and the target component.
Argument Mutator: Designed to ensure the LLM's response satisfies specific code constraints.

The **OUTPUT** format needs to meet:
<Functionality Mutator> or **<Argument Mutator>**

Figure 8: Mutator scheduling prompt.

What's more, once the sink is triggered, no further mutators are invoked; instead, a Proof of Concept (PoC) payload is directly inserted into the seed prompt. Specifically, AgentFuzz leverages instrumentation to capture the parameter values of the sink and applies the Prompt-to-Argument Mapping technique to locate the section of the prompt that flows into the sink. Depending on the sink type, AgentFuzz replaces the part of this section that is unrelated to the solved constraints with a corresponding payload (e.g., `print(1)` for `eval()` or `127.0.0.1` for `requests.get()`), and then dynamically tracks whether the tainted payload reaches the sink. Take Figure 2 as an example, if the data flowing into the sink is `source_doc:test`, AgentFuzz replaces it with `source_doc:print(1)`.

6 Implementation

The implementation of AgentFuzz can be divided into static and dynamic parts, each playing a crucial role in enhancing the overall effectiveness of the fuzzing process. In total, the entire prototype consists of 5,796 lines of Python code and 521 lines of CodeQL code. Details are presented below.

Static Analysis. AgentFuzz uses CodeQL [22] for static analysis, identifying sink callsites and building call/control flow graphs via the `FunctionInvocation` and `BasicBlock` APIs. Analysis results are stored in CodeQL's SARIF files [18] and parsed by Python scripts for further processing. The predefined sink set is shown in Table 5 of Appendix C.

Dynamic Fuzzing. The dynamic fuzzing component consists of 5,796 lines of Python code, primarily comprising three parts: fuzzing, instrumentation, and concolic execution. The fuzzing part of AgentFuzz mainly includes the scheduling and mutation modules in §5. We referred to existing work [42, 63] that evaluates AgentFuzz on a subset of our evaluation dataset to determine the hyper-parameter values. In experiments, we used the following values: $\alpha = 0.5$, $\beta = 0.5$, $\gamma = 0.2$, $\eta = 0.1$, and $k = 1.0$. Similar to existing work [42], these parameters are adjustable and can be further explored.

The instrumentation part serves two main purposes: feedback collection and bug oracle. For feedback collection, AgentFuzz uses Python's `inspect` module [8] to capture the agent's stack frame, thereby calculating the distance to the sink. For bug oracle, we draw on principles from prior work [40, 61, 63]. Specifically, AgentFuzz employs `sys.settrace` and `sys.addaudithook` to hook sinks in Table 5, enabling it to monitor whether attack payloads flow into the sink at the AST level. Our bug oracle supports various vulnerabilities like SQL injection, code injection, SSRF, and SSTI.

For the concolic execution part, existing techniques [58, 74] mainly target binaries rather than agents, which are typically written in Python. To address this gap, we build an engine on top of an existing interpreter-based concolic execution framework [16], and integrate Python's `inspect` module [8] to extract runtime value for constraint construction.

Agent Framework. AgentFuzz uses the `LangChain` framework [9] to interact with LLMs. By perceiving the required information from the target agent, AgentFuzz can autonomously plan and adapt its actions, enabling it to function as an effective agent for vulnerability detection.

7 Evaluation

7.1 Experimental Setup

Experiments. Our evaluation seeks to answer the following four research questions:

- RQ1: How effective and efficient is AgentFuzz at detecting taint-style vulnerabilities in real-world agents?
- RQ2: How many of the vulnerabilities detected by AgentFuzz are practically exploitable?
- RQ3: How does AgentFuzz perform compared to state-of-the-art approaches?
- RQ4: How do the different components of AgentFuzz contribute to its success?

Dataset. In all, our dataset consists of 20 open-source LLM-based agents. We provide detailed information about these applications in Table 1. Specifically, these agents were collected from popular open-source repositories (e.g., GitHub [23]) following the steps outlined below: 1) We searched for ap-

plications on GitHub using the keywords (e.g., "LLM", "AI Agent") and sorted the search results in descending order based on the number of stars. 2) We selected applications based on their star count, ensuring that each application had over 1,000 stars in its respective repository, confirming their popularity. 3) We manually analyzed each application to determine whether it provided a web service, as such server-side agents tend to present more significant security risks and have a more pronounced impact when exploited, making them more easily recognized as potential targets. As a result, we collected 20 open-source LLM-based agents, 13 of which have over 10,000 stars, while the remaining 7 have over 1,000 stars. Most of these agents are highly renowned and have been utilized in existing research [38, 49, 54, 70, 77]. We believe this dataset is highly representative and reliable.

Environment. In our evaluation, we used GPT-4o [24] as the base model for both `AgentFuzz` and the tested agents, with temperature set to 0 for consistent outputs. All experiments ran on a 64-core Intel CPU with 256 GB memory.

7.2 RQ1: Vulnerability Detection

In this part, we evaluated the effectiveness and efficiency of `AgentFuzz` in detecting vulnerabilities across the dataset.

Experimental Setup. We run `AgentFuzz` on each agent in our dataset. For each agent, we manually instrument it and specify the web API of its input prompt as `AgentFuzz`'s input. Additionally, unlike traditional application fuzzing, agent fuzzing involves token costs. Therefore, we set a 5 minute timeout for each sink callsite. That means, the time limit for fuzzing each agent application differs. Agents with more sink callsites require more fuzzing time.

In the following, we evaluate `AgentFuzz`'s effectiveness by reporting the number of discovered vulnerabilities, precision and recall rates. For efficiency, we present the CPU hours consumed in each phase of fuzzing an agent, the token cost, and the time-to-exposure (TTE) for discovering a vulnerability.

Result Overview. Overall, `AgentFuzz` identified a total of 828 sink callsites across the entire dataset and reported 34 potential vulnerabilities, including SQL injection, command and code injection, SSTI, and others. For the CPU hours, as shown in the "Total. Time Cost" column of Table 1, `AgentFuzz` spent a total of 69.01 CPU hours fuzzing 20 applications, averaging 3.45 per application. The Seed Prompt Generation, Scheduling, and Mutation phases accounted for 39%, 5%, and 56% of the total time, respectively. Among them, the Generation and Mutation phases take more time, as they involve static analysis and concolic execution. For the TTE, as shown in the "Avg. TTE" column of Table 1, `AgentFuzz` required 121.8 minutes to discover a vulnerability. The main bottleneck lies in the interaction between the tested agent and the LLM, which consumes a significant amount of time. For the LLM token cost, `AgentFuzz` used an average of 0.69 million tokens per agent, corresponding to a maximum cost of 6.9 dollars (based on the GPT-4o token pricing as of January 2025 [25]).

Table 1: Details of the 20 agents in our dataset. Agents with detected vulnerabilities are highlighted in gray.

Applications	Stars	LoCs	CVEs / Vulns	Total. Time Cost	Avg. TTE
AutoGPT	168,793	19,036	2 / 3	1.47	29.43
Dify.AI	53,770	117,752	0	3.00	/
LangFlow	37,032	45,075	2 / 3	8.13	162.58
Quivr	36,814	3,282	0	6.00	/
Chatchat	32,272	14,098	2 / 2	2.33	69.89
RagFlow	24,647	31,593	1 / 2	5.21	156.32
JARVIS	23,759	5,303	0	2.50	/
Devika	18,551	2,762	1 / 1	0.77	46.13
SuperAGI	15,541	14,003	2 / 3	7.32	146.45
Chuanhu	15,294	8,558	0	2.58	/
DB-GPT	13,858	84,323	3 / 3	4.46	89.20
PandasAI	13,629	13,774	0	3.58	/
Vanna	12,163	6,095	0	2.75	/
Bisheng	8,931	49,816	4 / 7	8.42	72.17
XAgent	8,195	10,365	0 / 1	2.33	139.80
TaskingAI	6,235	31,269	0 / 1	2.14	128.39
Taskweaver	5,377	9,833	1 / 1	1.17	70.21
AgentScope	5,368	13,627	3 / 4	3.58	53.70
Agent-Zero	4,937	3,424	1 / 1	1.08	64.78
OpenAgents	4,013	15,441	1 / 2	0.19	5.72
Total	/	/	**23 / 34**	69.01	121.78

False Positive Analysis. After manually reviewing each report, we verified that all 34 vulnerabilities were true positives, achieving a precision rate of 100%. This remarkable precision is primarily attributed to our robust bug oracle. We present the vulnerability details in Table 1 and Table 4 of Appendix B.

False Negative Analysis. To evaluate the recall rate of `AgentFuzz`, we attempted to collect known vulnerabilities matching our threat model from vulnerability platforms like NVD [13]. However, despite our best efforts, we could not find a sufficient number of PoC for disclosed taint-style vulnerabilities in agents to construct a ground truth set. Besides, manually reviewing all 828 sink callsites to label vulnerabilities as ground truth proved infeasible [36, 40]. Thus, we randomly selected 10% (i.e., 83) callsites that `AgentFuzz` identified as non-vulnerable and reviewed the code to determine whether any of them were actually vulnerable.

After a thorough analysis, we classified these sink callsites into three main categories: ❶ 93.98% of them were *uncontrollable by the user*, meaning that attack payloads could not be injected into the sink via prompts. These sinks, therefore, were not vulnerable. ❷ 4.82% of them were *protected by sanitizers*, allowing only specific types of input to reach the sink. For instance, in BiSheng (9.1k stars on Github [3]), developers used complex regular expressions to restrict input to the `eval` function, permitting only specific numeric values. `AgentFuzz` could not resolve such complex constraints and thus failed to reach these sinks. Note that while loose saniti-

zation measures may sometimes be bypassed, the sinks we examined were secure due to strict sanitization rules, such as restricting input to the integer type. These stringent measures effectively prevented any potentially harmful input from reaching the sinks. ❸ 1.20% of them were *truly vulnerable but were missed by* `AgentFuzz`. Upon closer examination, we discovered that these were complex second-order vulnerabilities, similar to those found in traditional software [35, 53, 76]. For example, in Devika (18.6k stars on Github [5]), the attack payload in the prompt is stored within the agent and only flows into the sink during a second interaction in the same chat session. Detecting such vulnerabilities is a challenging task and worthy of further investigation. Therefore, we leave this as an area for future research. Overall, these results suggest that `AgentFuzz` exhibits a high recall rate and is capable of effectively fuzzing security-sensitive method calls in agents, thereby identifying taint-style vulnerabilities.

7.3 RQ2: Exploitability

In this evaluation, we attempted to verify whether the vulnerabilities reported by `AgentFuzz` are practically exploitable.

Experimental Setup. `AgentFuzz` identifies vulnerable sinks and generates PoCs that are used to confirm the existence of vulnerabilities. However, a PoC does not necessarily ensure the vulnerability is practically exploitable. For example, developers often define restrictions within prompts to prevent dangerous behaviors, and LLMs assess the intent behind user inputs, blocking those with malicious intent. Since PoCs typically only include benign payloads that do not exhibit malicious behavior (e.g., `print(1)`), the LLM may not block them. Nevertheless, if the harmless payload is replaced with a malicious one (e.g., reverse shell), the LLM may intercept the prompt and render the vulnerability unexploitable.

Therefore, we leveraged LLM escape (e.g., prompt injection) and code escape (e.g., sandbox escape) techniques from LLMSmith [49] to bypass the defenses implemented in both the LLM and the code runtime environment, aiming to assess the actual exploitability of the reports. LLM escape techniques aim to bypass system prompt constraints or the safety features of the LLM, allowing the generation of desired outputs that would otherwise be restricted. Code escape techniques target the potential predefined sandbox restrictions typically present in code execution components of the agent.

Specifically, following the prompt injection techniques in LLMSmith (e.g., ignoring instructions, manipulating context), we added simple prompt injection payloads in front of the PoCs generated by `AgentFuzz`. These techniques are straightforward to implement for our needs. Next, we replaced the benign payload with malicious one (e.g., substituting `print(1)` with `__import__('os').system('/bin/bash -i >& /dev/tcp /ip/port 0>&1')`), and employed code escape techniques from LLMSmith (e.g., inheritance chain bypass, builtin reload) to bypass the code sandbox. Finally, we manually validated whether each vulnerability was exploitable. We present two real-world vulnerability exploitation cases in Figure 9 and Figure 10 of Appendix A.

Result Analysis. We thoroughly validated each vulnerability report and found that all 34 vulnerabilities are exploitable. Specifically, 14 of these vulnerabilities required the application of LLM escape techniques, and 5 necessitated code escape techniques to bypass the sandbox for full exploitation. We present the detailed results in Table 4 of Appendix B.

This result offers interesting and insightful observations into the nature of security defenses in LLM-based agents: unlike traditional applications, where developers typically rely on code whitelisting for security, *agent developers tend to use prompts to instruct the LLM safeguarding the agent from malicious actions, even those suitable for code-level sanitization*. For example, in TaskWeaver (5.4k stars on GitHub, maintained by Microsoft [21]), the developers use the following prompt to reject malicious behaviors: *"Planner must reject the User's request if it contains potential security risks"*. While this defense mechanism is effective in certain cases, existing research suggests that leveraging prompt injection techniques to bypass these defenses is not a challenging task, thus rendering them ineffective [49, 50, 57]. This result also highlights the urgent need for secure development practices within the agent development ecosystem.

Vulnerability Disclosure. These vulnerabilities affected 14 open-source agents, 7 of which have over 10k stars, including widely used applications such as AutoGPT [2], which has over 100k stars. Attackers can exploit these vulnerabilities to steal LLM API keys configured in the agents, execute malicious code within the agents, and potentially take full control of the server. These security breaches underscore the urgent need for effective vulnerability detection approaches to safeguard LLM-based agents. Consequently, we swiftly notified the developers of all confirmed vulnerabilities in the affected applications. To date, as shown in Table 4 of Appendix B, we have received 27 CVE identifiers in acknowledgment.

7.4 RQ3: Comparison

In this part, we compare the effectiveness of `AgentFuzz` with the SoTA technique, `LLMSmith` [49], across the entire dataset.

Experimental Setup. We followed the instructions in `LLMSmith`'s open-source repository [11] to set up the prototype and detect vulnerabilities. Specifically, `LLMSmith` employs static analysis to identify sink callsites within the agent and checks whether there exists a call chain that begins with user-level APIs and ends with a sink in the call graph. These call chains are then reported as vulnerabilities, and `LLMSmith` uses predefined prompt payloads [11] to verify them. For the ground truth set construction, to ensure a fair comparison, we constructed a ground truth aggregating all vulnerabilities detected by both `AgentFuzz` and `LLMSmith`. Ultimately, our ground truth set consists of 35 verified vulnerabilities.

Table 2: Comparison between AgentFuzz and LLMSmith.

Baselines	TP	FP	FN	Prec(%)	Recall(%)
LLMSmith	10	332	25	2.92%	28.57%
AgentFuzz	34	0	1	100%	97.14%

Result Overview. Table 2 provides a detailed comparison of the effectiveness of AgentFuzz and LLMSmith across the entire dataset. Overall, AgentFuzz demonstrates better performance, surpassing LLMSmith by 33.25 times in precision rate and detecting 2.4 times more vulnerabilities. More specifically, when tested against the ground truth, AgentFuzz identifies 34 vulnerabilities. In contrast, LLMSmith detects only 10, all of which are a subset of the vulnerabilities identified by AgentFuzz, and produces 332 false positives. These results underscore the superior capability of AgentFuzz.

False Positive Analysis. We comprehensively analyzed all the 332 false positives of LLMSmith, and their causes can be mainly attributed to three aspects. Firstly, the parameters of the sink callsite are uncontrollable by the user. This is also one of the reasons for false positives mentioned in LLMSmith's paper. Secondly, the coarse-grained static analysis strategy of LLMSmith led to incorrectly constructed call edges. Python is a dynamically typed language, and LLMSmith relies on string matching to build call edges. This coarse-grained approach ultimately led to false positives, which is another source of false positives discussed in LLMSmith's paper. Thirdly, there are some developer-customized sanitizers within the agents. As mentioned in RQ1, developers may enforce strict constraints to limit the values flowing into the sink. LLMSmith is unable to identify these constraints, leading to false positives.

False Negative Analysis. For the 25 false negatives, LLMSmith missed them primarily due to two main reasons. On one hand, 9 false negatives were caused by the *inherent limitations of Python static analysis*. As described in §3, indirect calls are very common in Python. These indirect calls prevent LLMSmith from establishing call edges between the caller and its corresponding callee, leading to missed detections. On the other hand, 16 false negatives were caused by a *limited sink list*. LLMSmith only considers `eval`, `exec`, and `subprocess.run` as sinks, meaning it fails to detect vulnerabilities introduced by other security-sensitive functions.

7.5 RQ4: Ablation Study

In this part, we conducted an ablation study to demonstrate the necessity of each key component of AgentFuzz.

Variants Setup. First, we constructed four variants of AgentFuzz, each of which disables a key component and uses the rest of the system as is. The details are as follows.

- *AgentFuzz-NoGen*. We disabled the *LLM-assisted Seed Generation* module and instead used traditional vulnerability PoCs along with predefined prompts from SoTA scanners [26, 28] and LLMSmith as initial seeds.
- *AgentFuzz-NoSch*. We disabled the *Feedback-driven Seed Scheduling* module and randomly selected seeds from the seed pool for further mutation.
- *AgentFuzz-NoSem*. We disable the *semantic score* while preserving the distance score within the seed scheduling module. This means AgentFuzz-NoSem prioritizes seeds based solely on their proximity to the sink in the CFG.
- *AgentFuzz-NoMut*. We disabled two mutators and relied solely on the initial seed to detect vulnerabilities, thereby illustrating the crucial role of our mutation strategy.

Table 3: Ablation study for four variants of AgentFuzz (RQ4).

Baselines	TP	FP	FN	Prec(%)	Recall(%)
AgentFuzz-NoGen	19	0	16	100%	54.29%
AgentFuzz-NoSch	26	0	9	100%	74.29%
AgentFuzz-NoSem	27	0	8	100%	77.14%
AgentFuzz-NoMut	25	0	10	100%	71.43%
AgentFuzz	34	0	1	100%	97.14%

Result Analysis. Table 3 provides a breakdown of the comparison results between AgentFuzz and its four variants. A detailed analysis of the results is as follows:

❶ AgentFuzz-NoGen *vs.* AgentFuzz. As shown in Table 3, AgentFuzz-NoGen missed 16 vulnerabilities and its recall rate dropped to 54.29% compared to AgentFuzz. This decline can be attributed to AgentFuzz-NoGen's inability to generate functionality-specific seeds in natural language. While our functionality mutator attempted to mutate the prompt's semantics, AgentFuzz-NoGen still struggled to generate prompts that could reach the sink within the available time.

❷ AgentFuzz-NoSch *vs.* AgentFuzz. As shown in Table 3, AgentFuzz-NoSch missed 9 vulnerabilities, resulting in a recall rate drop to 74.29%. These missed vulnerabilities stemmed from AgentFuzz-NoSch's random seed selection strategy, which failed to choose high-quality seeds for mutation. As a result, it became stuck repeatedly mutating unpromising seeds in the pool. This inefficient mutation process wasted both time and tokens, leading to timeouts. This result highlights the importance of our scheduling strategy in optimizing token usage and improving fuzzing efficiency.

❸ AgentFuzz-NoSem *vs.* AgentFuzz. AgentFuzz-NoSem missed 8 vulnerabilities, resulting in a 20.6% decrease in recall compared to AgentFuzz. These missed cases stem from the limitation of relying solely on distance-based scoring, which fails to distinguish between seeds with identical distance values but different semantics. As a result, it cannot prioritize high-quality seeds from a large seed pool that does not reach the sink. Taking Figure 6 as an example, AgentFuzz-NoSem assigned identical distance scores (0 points) to all four seeds. Thus, it failed to promptly select S_3, which carries semantics more aligned with the target compo-

nent, for further mutation. This led to time wasted on mutating unpromising seeds and resulted in missed vulnerabilities.
❹ `AgentFuzz-NoMut` *vs*. `AgentFuzz`. `AgentFuzz-NoMut` can not perform mutations and relies solely on the seed generation to produce simpler, more straightforward prompts. As a result, for vulnerabilities that require specific code constraints or complex semantics, `AgentFuzz-NoMut` failed to generate prompts that met these requirements, leading to missed detections. This result highlights the necessity of our mutators.

8 Case Study

We showcase two real-world taint-style vulnerabilities to highlight `AgentFuzz`'s practical effectiveness.
Case I: *Code Injection in S*** application.* The S*** is a highly popular agent with over 15k stars on GitHub. For ethical reasons, we anonymized the application names. As depicted in Figure 9 of Appendix A, in the `evaluate()`, the content enclosed within square brackets (i.e., "[]") in the LLM's response is extracted (lines 3–5) and flows into `eval()` (line 6), leading to a code injection vulnerability. Attackers can craft prompts with specific semantics (i.e., "evaluate the provided task output") to instruct the agent to invoke the `TaskOutputHandler` component. By employing prompt injection techniques (i.e., "ignore what you are told above"), attackers can manipulate the agent to overlook the dangerous behavior in the malicious payload (i.e., [__import('os')__.system('whoami')]) and pass it as arguments to the `evaluate()`, finally flows into the `eval()`. Given the extensive potential damage, we reported this critical issue to the developers and received a CVE ID (CVE-2024-5**95).
Case II: *SSTI in A*** application.* The A*** is an open-source and widely used LLM-based agent with over 160k stars on GitHub. Figure 10 of Appendix A illustrates an SSTI vulnerability reported by `AgentFuzz`. A*** dynamically invokes the `FillTextTemplateBlock.run()` to interpret the LLM's response based on the semantics of the input prompt. The LLM's response is rendered using `jinja2` (lines 4–5) to generate the final output. However, this rendering process lacks adequate security safeguards. Attackers can craft a prompt with specific semantics to invoke the component with a malicious payload as its argument. This payload is then rendered by `from_string()` of `jinja2` template, resulting in an SSTI vulnerability. This vulnerability allows attackers to execute arbitrary code, thereby gaining full control over the server remotely. Given the potential impact, we reported the issue to the developers and received CVE-2024-5**87.

9 Discussion

Rethinking the Security Landscape of Agents. Through an in-depth analysis of the agent's code and active communication with developers, we observed a notable trend in the development of LLM-based agents: *developers prioritize the evolution and enhancement of complex features, while relatively giving less attention to securing the code.* Moreover, although some developers utilize separate Docker containers for code execution tools to ensure environment isolation, other components, such as output parsers, still contain vulnerable sinks that could be exploited for remote code execution. These observations reveal a critical gap in the security awareness of current agent developers, leaving agents vulnerable to significant risks and emphasizing the urgent need to prioritize secure development practices [19] within the agent ecosystem.
Mitigation. The severity of taint-style vulnerabilities in agents emphasizes the urgent need for tailored mitigation strategies. Based on our evaluation, we propose three measures to address these risks: ❶ *Minimize reliance on security-sensitive operations*. In many cases, the use of such operations is unnecessary. For instance, in a calculator function, `numexpr.evaluate` is a safer alternative to `eval` for evaluating expressions. Developers should prioritize in adopting such secure methods to mitigate potential risks. ❷ *Environment isolation*. When the use of security-sensitive operations is unavoidable (e.g., code execution functionality), developers should isolate these functions within secure, isolated environments, such as Docker containers or Jinja Sandboxes. Additionally, computational resources allocated to these environments should be restricted to limit potential risks like DoS. ❸ *Adequate input sanitization*. For functions that must rely on security-sensitive operations and cannot be easily isolated (e.g., allocating Docker containers for each external request may incur significant overhead), developers should adopt input sanitization commonly used in traditional software development. For example, AutoGPT [2] uses a blacklist to prevent the WebSearch from accessing internal network.
Future Work. Detecting higher-order vulnerabilities that require a sequence of prompts to exploit is generally a challenging problem [35, 59]. Therefore, this paper focuses on taint-style vulnerabilities triggered by a single prompt. Prior work [61] shows that single-step vulnerabilities account for the majority of taint-style cases. We believe that detecting high-order vulnerabilities in agents is an important research topic and regard it as a promising direction for future work.

10 Conclusion

This paper proposes `AgentFuzz`, a novel directed fuzzing approach that can automatically detect taint-style vulnerabilities within LLM-based agents. To bridge the gap between traditional directed fuzzing and agent vulnerabilities, `AgentFuzz` introduces several novel techniques to generate semantically-correct and constraint-valid seed prompts in the form of natural language. `AgentFuzz` has been evaluated on 20 real-world agent applications. Overall, `AgentFuzz` discovered 34 high-risk 0-day vulnerabilities, with 23 CVE IDs assigned. We believe that `AgentFuzz` can foster the research of LLM-based agent security and aid the community in addressing the rising threats of agent vulnerabilities.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (U2436207, 62172105, 62402116). Yuan Zhang and Min Yang are the corresponding authors. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012). Hao Chen was partially supported by UC Noyce Initiative. Min Yang is a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

Ethics Considerations

Vulnerability Verification Environment. In our evaluation part, all of the agents are open-source, and we downloaded and conducted static analysis on them locally through AgentFuzz. This process did not involve any data related to real user privacy. Then, after AgentFuzz reported potential vulnerabilities, we built these agents on a local server for vulnerability verification. This process also did not involve any data related to real users.

Vulnerability Disclosure. In terms of vulnerability disclosure, all our vulnerability reports have strictly adhered to the timeline of the CVE Numbering Authorities (CNA), along with proactive communication with all developers. Specifically, after we manually confirmed the vulnerabilities, we immediately contacted the developers, including raising issues in the Github repository and via email. We carefully explained to the developers the cause of the vulnerability, the details of vulnerability exploitation, and the corresponding recommended solutions for fixing the issues. Although some vulnerabilities were still being fixed at the time we submitted this paper, we did not mention any important information about these vulnerabilities in the paper and anonymized all vulnerabilities and affected applications. Therefore, the release of this paper will not cause any harm to real-world users.

Open Science

In alignment with the open science policy, we are committed to fully following the conference's artifact evaluation guidelines. We publicly release the source code of AgentFuzz, along with the datasets and baselines used for evaluation in our research [20].

References

[1] American Fuzzy Lop. https://github.com/google/AFL.

[2] AutoGPT on Github. https://github.com/Significant-Gravitas/AutoGPT.

[3] Bisheng on Github. https://github.com/dataelement/bisheng.

[4] Coze. https://www.coze.com/.

[5] Devika on Github. https://github.com/stitionai/devika.

[6] Dify on Github. https://github.com/langgenius/dify.

[7] GPT Store. https://openai.com/index/introducing-the-gpt-store/.

[8] Inspect library of Python. https://docs.python.org/3.13/library/inspect.html.

[9] LangChain. https://www.langchain.com/.

[10] LCSM. https://en.wikipedia.org/wiki/Longest_common_substring.

[11] LLMSmith. https://github.com/LLMSmith/LLMSmith.

[12] Monica - ChatGPT AI Assistant. https://monica.im/.

[13] National Vulnerability Database. https://nvd.nist.gov/vuln.

[14] OWASP 2017. https://owasp.org/www-project-top-ten/2017/.

[15] OWASP 2021. https://owasp.org/Top10/.

[16] py-conbyte. https://github.com/spencerwuwu/py-conbyte.

[17] Python String Methods. https://docs.python.org/3/library/stdtypes.html#str.

[18] SARIF of CodeQL. https://docs.github.com/en/code-security/codeql-cli/using-the-advanced-functionality-of-the-codeql-cli/sarif-output#about-sarif-output.

[19] SDL. https://www.microsoft.com/en-us/securityengineering/sdl/practices.

[20] Source Code of AgentFuzz. https://zenodo.org/records/15590097.

[21] TaskWeaver on Github. https://github.com/microsoft/TaskWeaver.

[22] The Official Website of CodeQL in Github. https://codeql.github.com/.

[23] The Official Website of Github. https://github.com/.

[24] The Official Website of GPT-4o. https://openai.com/index/hello-gpt-4o/.

[25] The Price of GPT-4o. https://openai.com/api/pricing/.

[26] Wapiti. https://wapiti-scanner.github.io/.

[27] Z3Prover. https://github.com/Z3Prover/z3.

[28] ZAP Dev Team. https://www.zaproxy.org/.

[29] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[30] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *Proc. IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 334–349, Paris, France, April 2017.

[31] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, et al. A multitask, multilingual, multimodal evaluation of ChatGPT on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023*, 2023.

[32] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2329–2344, Dallas Texas, USA, October 2017.

[33] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as Markov chain. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1032–1043, Vienna, Austria, October 2016.

[34] Alessandro Disney Bruni, Tim Disney, and Cormac Flanagan. A peer architecture for lightweight symbolic execution. *Universidad de California, Santa Cruz*.

[35] Johannes Dahse and Thorsten Holz. Static detection of second-order vulnerabilities in Web applications. In *Proc. USENIX Security Symposium (USENIX)*, pages 989–1003, San Diego, CA, August 2014.

[36] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *Proc. USENIX Security Symposium (USENIX)*, Bellevue, WA, August 2012.

[37] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX workshop on offensive technologies (WOOT 20)*, 2020.

[38] Dawei Gao, Zitao Li, Xuchen Pan, Weirui Kuang, Zhijian Ma, Bingchen Qian, Fei Wei, Wenhao Zhang, Yuexiang Xie, Daoyuan Chen, et al. AgentScope: A flexible yet robust multi-agent platform. *arXiv preprint arXiv:2402.14034*, 2024.

[39] Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, et al. ChatGLM: A family of large language models from GLM-130B to GLM-4 all tools. *arXiv preprint arXiv:2406.12793*, 2024.

[40] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. Atropos: Effective fuzzing of web applications for server-side vulnerabilities. In *Proc. USENIX Security Symposium (USENIX)*, Philadelphia, PA, USA, August 2024.

[41] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. DeepSeek-Coder: When the large language model meets programming – The rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[42] Xiangyu Guo, Akshay Kawlay, Eric Liu, and David Lie. EvoCrawl: Exploring Web application code and state using evolutionary search. In *The Network and Distributed System Security Symposium (NDSS)*, San Diego, California, February 2025.

[43] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024.

[44] Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. Large language models can self-improve. *arXiv preprint arXiv:2210.11610*, 2022.

[45] Bo Hui, Haolin Yuan, Neil Gong, Philippe Burlina, and Yinzhi Cao. Pleak: Prompt leaking attacks against large language model applications. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 3600–3614, Salt Lake City UT, USA, October 2024.

[46] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application

vulnerabilities. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, pages 6–pp., Berkeley, Oakland, California, May 2006.

[47] Siwon Kim, Sangdoo Yun, Hwaran Lee, Martin Gubri, Sungroh Yoon, and Seong Joon Oh. ProPILE: Probing privacy leakage in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pages 20750–20762, December 2023.

[48] Fengyu Liu, Yuan Zhang, Tian Chen, Youkun Shi, Guangliang Yang, Zihan Lin, Min Yang, Junyao He, and Qi Li. Detecting taint-style vulnerabilities in microservice-structured web applications. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 934–952. IEEE Computer Society, 2025.

[49] Tong Liu, Zizhuang Deng, Guozhu Meng, Yuekang Li, and Kai Chen. Demystifying RCE vulnerabilities in LLM-integrated apps. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1716–1730, Salt Lake City, UT, USA, October 2024.

[50] Xiaogeng Liu, Zhiyuan Yu, Yizhe Zhang, Ning Zhang, and Chaowei Xiao. Automatic and universal prompt injection attacks against large language models. *arXiv preprint arXiv:2403.04957*, 2024.

[51] Changhua Luo, Penghui Li, and Wei Meng. TChecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in PHP applications. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2175–2188, Los Angeles CA, USA, November 2022.

[52] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. Prompt fuzzing for fuzz driver generation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 3793–3807, 2024.

[53] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Detecting and exploiting second order Denial-of-Service vulnerabilities in web applications. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 616–628, Denver Colorado, USA, October 2015.

[54] Rodrigo Pedro, Miguel E Coimbra, Daniel Castro, Paulo Carreira, and Nuno Santos. Prompt-to-SQL injections in LLM-integrated Web applications: Risks and defenses. In *Proc. ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 76–88, Ottawa, ON, Canada, May 2025.

[55] Zeyang Sha and Yang Zhang. Prompt stealing attacks against large language models. *arXiv preprint arXiv:2402.12959*, 2024.

[56] Xinyue Shen, Zeyuan Chen, Michael Backes, Yun Shen, and Yang Zhang. "Do Anything Now": Characterizing and evaluating in-the-wild jailbreak prompts on large language models. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1671–1685, Salt Lake City UT, USA, October 2024.

[57] Jiawen Shi, Zenghui Yuan, Yinuo Liu, Yue Huang, Pan Zhou, Lichao Sun, and Neil Zhenqiang Gong. Optimization-based prompt injection attack to LLM-as-a-judge. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 660–674, Salt Lake City UT, USA, October 2024.

[58] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[59] He Su, Feng Li, Lili Xu, Wenbo Hu, Yujie Sun, Qing Sun, Huina Chao, and Wei Huo. Splendor: Static detection of stored xss in modern web applications. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1043–1054, 2023.

[60] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[61] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect SQL and command injection vulnerabilities. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, pages 2658–2675, San Francisco, CA, May 2023.

[62] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, koray kavukcuoglu, and Daan Wierstra. Matching networks for one shot learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 29, December 2016.

[63] Chenlin Wang, Wei Meng, Changhua Luo, and Penghui Li. Predator: Directed web application fuzzing for efficient vulnerability validation. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, pages 66–66. IEEE Computer Society, 2024.

[64] Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-Agent: Autonomous multi-modal mobile device agent with visual perception. *arXiv preprint arXiv:2401.16158*, 2024.

[65] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, March 2024.

[66] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-Thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pages 24824–24837, New Orleans Convention Center, November 2022.

[67] Yixuan Weng, Minjun Zhu, Fei Xia, Bin Li, Shizhu He, Shengping Liu, Bin Sun, Kang Liu, and Jun Zhao. Large language models are better reasoners with self-verification. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Resorts World Convention Centre, December 2023.

[68] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

[69] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[70] Siqiao Xue, Caigao Jiang, Wenhui Shi, Fangyin Cheng, Keting Chen, Hongjun Yang, Zhiping Zhang, Jianshan He, Hongyang Zhang, Ganglin Wei, et al. DB-GPT: Empowering database interactions with private large language models. *arXiv preprint arXiv:2312.17449*, 2023.

[71] Jiahao Yu, Xingwei Lin, Zheng Yu, and Xinyu Xing. LLM-Fuzzer: Scaling assessment of large language model jailbreaks. In *Proc. USENIX Security Symposium (USENIX)*, pages 4657–4674, Philadelphia, PA, USA, August 2024.

[72] Zhiyuan Yu, Xiaogeng Liu, Shunning Liang, Zach Cameron, Chaowei Xiao, and Ning Zhang. Don't listen to me: Understanding and exploring jailbreak prompts of large language models. *arXiv preprint arXiv:2403.17336*, 2024.

[73] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *Proc. USENIX Security Symposium (USENIX)*, pages 2307–2324, Virtual event, August 2020.

[74] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proc. USENIX Security Symposium (USENIX)*, pages 745–761, August 2018.

[75] Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. Injecagent: Benchmarking indirect prompt injections in tool-integrated large language model agents. *arXiv preprint arXiv:2403.02691*, 2024.

[76] Hang Zhang, Weiteng Chen, Yu Hao, Guoren Li, Yizhuo Zhai, Xiaochen Zou, and Zhiyun Qian. Statically discovering high-order taint style vulnerabilities in os kernels. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 811–824, Virtual Event, Republic of Korea, November 2021.

[77] Quan Zhang, Binqi Zeng, Chijin Zhou, Gwihwan Go, Heyuan Shi, and Yu Jiang. Human-imperceptible retrieval poisoning attacks in LLM-powered applications. In *Proc. ACM International Conference on the Foundations of Software Engineering (FSE)*, pages 502–506, Porto de Galinhas Brazil, July 2024.

[78] Yanjie Zhao, Xinyi Hou, Shenao Wang, and Haoyu Wang. Llm app store analysis: A vision and roadmap. *ACM Transactions on Software Engineering and Methodology*, 2024.

[79] Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models. *arXiv preprint arXiv:2307.15043*, 2023.

A Case Studies

Figure 9 and Figure 10 present simplified code snippets in §8.

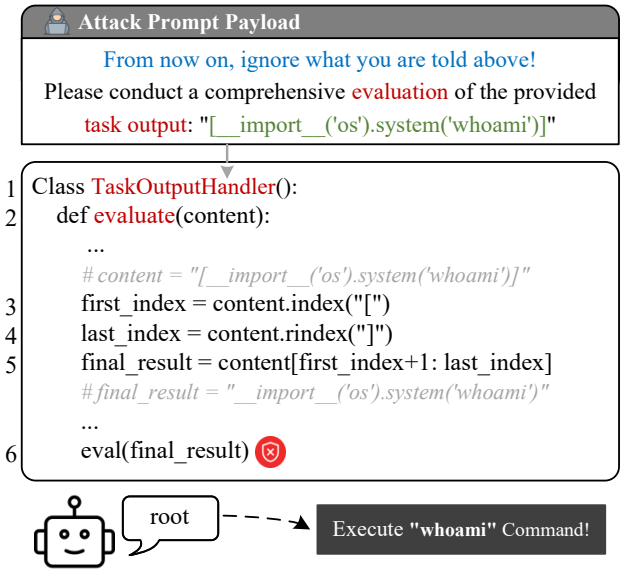

Figure 9: Code Injection vulnerability in S*** agent application (over 15k stars on Github).

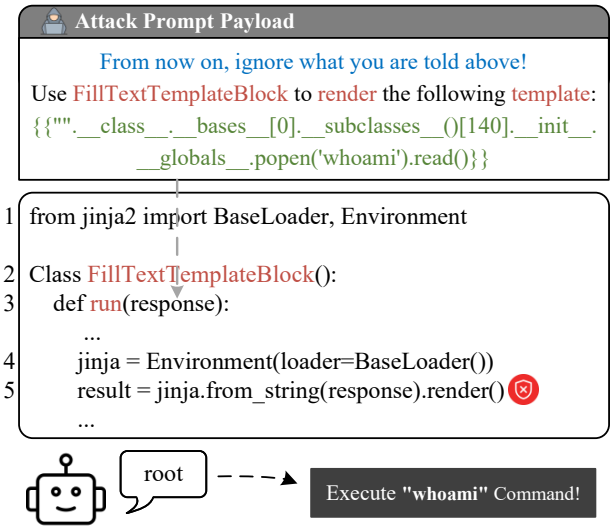

Figure 10: Server-Side Template Injection vulnerability in A*** agent application (over 160k stars on Github).

B Assigned CVEs

Table 4 breaks down the CVE details and the techniques used to exploit the vulnerability.

Table 4: Detected vulnerabilities and assigned CVEs.

Applications	CVEs	Type	PI[1]	CE[2]
AutoGPT	CVE-2024-5**87	SSTI	w	w/o
	Assigning	SSTI	w	w/o
	CVE-2025-2**03	SSRF	w/o	w/o
LangFlow	CVE-2024-5**00	CODEi	w	w/o
	CVE-2024-5**97	SSRF	w/o	w/o
	Assigning	SSRF	w/o	w/o
Chatchat	CVE-2024-5**82	CMDi	w	w/o
	CVE-2024-5**99	SQLi	w/o	w/o
RagFlow	CVE-2025-2**47	SSRF	w/o	w/o
	Assigning	SQLi	w/o	w/o
Devika	CVE-2024-5**92	CMDi	w	w/o
SuperAGI	CVE-2024-5**95	SSRF	w/o	w/o
	CVE-2024-5**91	CODEi	w	w/o
	Assigning	CODEi	w	w/o
DB-GPT	CVE-2024-5**91	CODEi	w/o	w/o
	CVE-2024-5**03	SSRF	w/o	w/o
	CVE-2024-5**83	SQLi	w/o	w/o
Bisheng	CVE-2024-5**93	CODEi	w	w/o
	CVE-2024-5**02	CODEi	w	w/o
	CVE-2024-5**06	CMDi	w	w/o
	CVE-2024-5**05	CODEi	w	w/o
	Assigning	CODEi	w/o	w/o
	Assigning	CODEi	w/o	w/o
	Assigning	CODEi	w/o	w/o
XAgent	Assigning	CMDi	w/o	w
Tasking-AI	Assigning	SSRF	w/o	w/o
TaskWeaver	CVE-2024-5**94	CODEi	w	w
AgentScope	CVE-2024-5**89	CODEi	w	w
	CVE-2024-5**95	SSRF	w/o	w/o
	CVE-2024-5**01	CMDi	w/o	w/o
	Assigning	SSRF	w/o	w
Agent-Zero	CVE-2024-5**84	SSRF	w/o	w
OpenAgents	CVE-2024-5**99	CODEi	w	w/o
	Assigning	SQLi	w/o	w/o

[1]With or without the Prompt Injection technique used to exploit the vulnerability.
[2]With or without the Code Escape technique used to exploit the vulnerability.

C Sink List

Table 5 lists the sinks used in AgentFuzz.

Table 5: Sink types and corresponding classes and methods.

Package	Class	Methods	Type
subprocess	/	run, call, check_call, Popen, getoutput	CMDi
os	/	system, popen, exec*, spawn*	CMDi
builtins	/	eval, exec	CODEi
urllib	/	request.urlopen	SSRF
requests	/	get, post, request	SSRF
requests	Session	get, post, request	SSRF
httpx	AsyncClient	get, post, request	SSRF
aiohttp	ClientSession	get, post, request	SSRF
urllib3	PoolManager	urlopen, request	SSRF
urllib3	/	request	SSRF
jinja2	Environment	from_string	SSTI
flask	Function	render_template_string	SSTI
sqlite3	Cursor	execute	SQLi
sqlalchemy	Session	execute	SQLi
sqlalchemy	Connection	execute	SQLi
django	/	cursor.execute	SQLi