# Detecting Taint-Style Vulnerabilities in Microservice-Structured Web Applications

Fengyu Liu†, Yuan Zhang†, Tian Chen†, Youkun Shi†, Guangliang Yang†, Zihan Lin†, Min Yang†, Junyao He‡, Qi Li‡
*Fudan University†, Alibaba Group‡*

***Abstract*—Microservice architecture has been becoming increasingly popular for building scalable and maintainable applications. A microservice-structured web application (shortened to microservice application) enhances security by providing a loose-coupling design and enforcing the security isolation between different microservices. However, in this paper, our study shows microservice applications still suffer from taint-style vulnerability, one of the most serious vulnerabilities. We propose a novel security analysis approach, named `MScan`, that can effectively detect taint-style vulnerabilities in real-world evolving-fast microservice applications. Our approach mainly consists of three phases. First, `MScan` identifies the entry points accessible to external malicious users by applying a gateway-centric analysis. Second, `MScan` utilizes a new data structure, i.e. service dependence graph, to bridge inter-service communication. Finally, `MScan` employs a distance-guided strategy for selective context-sensitive taint analysis to detect vulnerabilities. By applying `MScan` on 25 open-source microservice applications and 5 industrial microservice applications from a world-leading fintech company, we found `MScan` can effectively vet these applications with the discovery of 59 high-risk 0-day vulnerabilities. We have conducted responsible vulnerability disclosure. Up to now, 31 CVE identifiers have been issued.**

## 1. Introduction

Microservice architecture has recently become increasingly popular in the web area, for its technical benefits of building scalable and maintainable web applications [21, 22]. It differs from traditional monolithic architecture, which faces difficulties in handling modern evolving-fast web applications. Instead, microservice architecture significantly improves scalability, simplifies updates and maintenance, and facilitates continuous release and deployment capabilities. Many giant companies, like Uber [10], X/Twitter [9], and Amazon [8], have embraced microservice designs to build their sophisticated web platforms.

From the perspective of system security, a microservice application enables improved fault and security isolation. It fosters the creation and integrity of independent services (i.e. *microservice* components) that run in separate processes and communicate through lightweight mechanisms. Such a loose-coupling and isolation design helps maintain clear boundaries and shrink the attack surface compared to monolithic applications, where modules might share resources and have more direct access to each other.

As a result, there is a common perception that microservice applications are inherently more secure and less prone to breaches. Nevertheless, we find this notion does not fully capture the complexities of the microservice security landscape. We find microservice applications are not completely immune to security issues, and can still be susceptible to taint-style vulnerability, which is one of the most serious security flaws and ranked as the top web vulnerability by OWASP [13, 14]. This is because a microservice application is commonly manifested as a web server and is designed to be responsive to user quests. Attackers can leverage this responsiveness, exploit the user-accessible entry points exposed by the gateway and inter-service communication mechanisms, and finally abuse the application's sensitive functionalities.

In this work, we aim to design a vulnerability detection approach that can effectively vet the security of real-world popular Java-based microservice applications against taint-style vulnerabilities. Taint analysis has proven to be an effective technique for detecting such vulnerabilities in traditional monolithic applications [51]. However, existing techniques have not been specifically designed for microservice-structured web applications. They faced difficulties in being applied directly in this context. Specifically, three main challenges are encountered and should be carefully addressed.

- *C1: How to infer the user-accessible entry points by understanding the flexible semantics of the gateway configuration?* A microservice application consists of numerous microservices, each with its own communication entry points. Some of these entry points are exposed and accessible to users, while others can only be accessed internally. If the user-accessible entry points are not accurately identified, it could negatively impact the detection effectiveness. However, this is not a trivial task. The challenge lies in the unstructured content of routing rules, which complicates their interpretation and modeling.
- *C2: How to accurately establish connections between diverse inter-service communication senders and their corresponding receivers?* Inaccurate connections between senders and receivers during inter-service communication can lead to significant false negatives. The primary challenge in this analysis arises from the diverse im-

plementation mechanisms and diagrams for inter-service communication in real-world microservice applications. Identifying the sender among numerous method calls and accurately linking it to the corresponding receiver, which is often located in another service, presents substantial challenges.
- *C3: How to conduct effective taint analysis for detecting microservices vulnerabilities?* In the security analysis of microservices, achieving high precision is crucial for identifying high-value vulnerabilities that are often hidden deeply within the system. However, existing precise techniques, such as context-sensitive analysis, face significant limitations. In microservice applications, the call chains can be extremely long due to the concatenation of multiple services, making context-sensitive analyses highly resource-intensive. This often results in memory issues and timeout, creating barriers to effective vulnerability detection.

In this paper, we propose a novel security analysis approach for detecting taint-style vulnerabilities in microservice applications, called `MScan`. Our `MScan` approach consists of three primary phases. In the first phase, `MScan` conducts an LLM-assisted approach to understand the semantics of the gateway configuration file and subsequently identifies the user-accessible entry points. In the second phase, `MScan` identifies inter-service communication mechanisms with a novel data structure, called Service Dependence Graph (SDG). SDG helps conduct a comprehensive analysis of taint propagation across different microservice components. In the third phase, `MScan` employs a distance-guided strategy, which dynamically adjusts the level of context sensitivity based on the proximity of the analyzed method to the source-to-sink path within the call graph. Leveraging these techniques, `MScan` can effectively perform inter-service context-sensitive taint analysis to detect taint-style vulnerabilities within microservice applications.

To demonstrate the effectiveness and performance of `MScan`, we evaluated it on 25 widely-used open-source microservice applications and 5 industrial microservice applications. Our evaluation shows `MScan` identified 59 0-day taint-style vulnerabilities, achieving a precision rate of 71.95%. These newly discovered microservice vulnerabilities can be exploited to gain control over the entire application servers, and potentially result in significant financial losses. Considering the substantial security impact of these vulnerabilities, we responsibly reported them to the developers of the affected applications. As of now, 31 of these vulnerabilities have been assigned CVEs.

In summary, the paper makes the following main contributions:

- We conduct a systematic study on microservice security and understand its root cause of suffering taint-style vulnerabilities.
- We propose a novel taint-style vulnerability detection approach, called `MScan`, that can effectively vet the security of real-world microservice applications. We will open-source our `MScan` prototype upon publication.

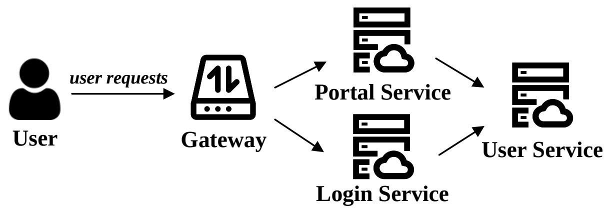


Figure 1: An example that illustrates the typical architecture of a microservice application.

- We evaluated `MScan` on 25 real-world popular microservice applications and 5 industrial microservice applications. `MScan` successfully identified 59 (confirmed) 0-day vulnerabilities. As of now, these vulnerabilities have been assigned 31 CVE identifiers.

## 2. Background

As web applications evolve, more and more are adopting a microservice architecture, such as Uber [10], Twitter [9], and Amazon [8]. Compared to traditional monolithic applications [25], microservice-structured web applications (shortened to *microservice application*) offer advantages in scalability and maintainability [21, 22]. Furthermore, microservice architecture provides lightweight security isolation between different service components. However, microservice applications are still particularly vulnerable to taint-style vulnerabilities, which can pose significant security risks.

In this section, we present an overview of the common architecture of microservice applications (in §2.1) and introduce the taint-style vulnerabilities within them (in §2.2).

### 2.1. Microservice-structured Web Application

A microservice-structured web application usually contains three key components: microservices, gateway, and inter-service communication. Below we present more of their details.

- *Microservices.* Microservice components are autonomous, self-contained services that collaborate seamlessly to collectively achieve complex functionalities. Each microservice is designed to perform its specific function independently, running in its own environment and communicating with other microservices within the same application through a well-defined API. For instance, as depicted in Figure 1, the user login feature on a portal website is facilitated through the synergistic operation of multiple microservices.
- *Inter-service Communication.* This refers to the manner in which microservices interact with each other. It can occur through various protocols and message formats, such as REST/HTTP [16] or gRPC [5]. This is crucial for ensuring that services can collaboratively perform complex tasks while maintaining loose coupling and high cohesion. As depicted in Figure 1, the `Login service`

retrieves user data from the `User service` through inter-service communication mechanisms, facilitating the login process.

- *Gateway.* This acts as the central entry point for all incoming requests in a microservice architecture, responsible for routing requests to the appropriate microservices [36]. It enforces strict routing policies, ensuring only allowed requests can be forwarded to the appropriate microservices, i.e., user-accessible services. As illustrated in Figure 1, the `Gateway` forwards requests for the internal microservices (i.e., `Portal` and `Login`) according to the routing rules specified in its configuration file. Meanwhile, it discards requests that attempt to directly access the `User service`, thereby preventing illicit access by malicious users.

## 2.2. Taint-style Vulnerability

Taint-style vulnerabilities are a widespread type of flaw in web applications and consistently rank high on the OWASP Top Ten list [13, 14]. These vulnerabilities typically occur when user-controlled variables are passed into security-sensitive functions without proper validation, potentially leading to critical exploits such as SQL Injection (SQLi), Server Side Request Forgery (SSRF), and XML External Entity Injection (XXE).

Given the inherent characteristics of the architecture, taint-style vulnerabilities in microservice applications can be divided into two categories based on their detection and exploitation differences: intra-service vulnerability and inter-service vulnerability.

**Intra-service Vulnerability.** Taint-style vulnerabilities of the intra-service type typically occur within a single service of an application. As illustrated in Figure 2, the `@Path` annotation on line 1 signifies that this endpoint can be accessed by users through the URL "/portal/query". Users can query specific values by inputting the "id" parameter to the `select()` method (line 4). However, as the user-controllable "id" parameter is passed to the `eval()` method without undergoing any security validation, it presents a potential security flaw. A malicious attacker could exploit this by inputting a carefully crafted malicious input, using the `eval()` method to execute arbitrary commands on the server.

```
  // Portal Service (can be accessed)
1 @Path(value = "/portal/query")
2 public User query(String id) {
    ...
3   String query = (new ScriptEngineManager()).eval(id);
4   return select(query);
5 }
```

Figure 2: An example of intra-service vulnerability in microservice application.

**Inter-service Vulnerability.** Taint-style vulnerabilities in the inter-service type manifest within multiple services of an application. Unlike intra-service vulnerabilities, each microservice involved does not present a security threat on its own. However, due to the data exchange occurring through inter-service communication between them, a latent security risk is engendered. Figure 3 is used as an illustrative example to shed light on the root cause behind inter-service vulnerabilities within a microservice application. It depicts the collaboration between two services - the `Portal` service and the `User` service - both of which are deployed on separate machines within a microservice application. The `Portal` service dynamically generates tasks and sends them to the Kafka [6] message queue (line 4), thereby invoking the user information querying function in the `User` service (line 11). When viewed in isolation, neither of these services poses any security risk. The `Portal` service lacks the necessary sinks that could be exploited, and the `User` service is devoid of a controllable source. Nevertheless, the scenario takes an interesting turn due to the existence of inter-service communication. A security vulnerability emerges when these two services work in unison. Once an attacker gains insight into the data communication mechanisms of the two services, they can exploit this vulnerability from the user-accessible service (i.e., entry point) to the vulnerability service. Specifically, this can be done by injecting a meticulously crafted malicious input into the Kafka message queue within the `Portal` service. This input is subsequently retrieved in the `eval` method within `User` service, leading to the triggering of the vulnerability.

```
   // Portal Service (can be accessed)
 1 @Path(value = "/portal/query")
 2 public User query(String id) {
 3   String op = "query";
 4   KafkaProducer kafkaProducer =
            new KafkaProducer(String.format("user/%s", op));
 5   kafkaProducer.send(id);
     ...
 6 }

   // User Service (can NOT be accessed)
 7 @KafkaListener(topics="user/query")
 8 public User queryTask() {
     …
 9   String id = kafkaConsumer.poll();
10   String query = (new ScriptEngineManager()).eval(id);
11   return select(query);
12 }
```

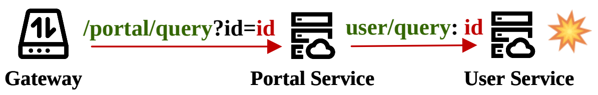


Figure 3: An example of inter-service vulnerability in microservice application.

## 3. Challenges and Solutions

According to Jetbrains' survey report [21, 22], Java is the most commonly used language for developing microservice applications (e.g., Uber [10], Twitter [9] or Amazon [8]). To the best of our knowledge, there are no existing static analysis techniques specifically designed for detecting taint-style vulnerabilities in Java-based microservice applications. To achieve this, an intuitive and straightforward workflow can be outlined as follows: ❶ Understand the gateway configuration files to determine the user-accessible entry point, i.e., the source; ❷ Analyze inter-service communication mechanisms to establish data flow connections between microservices; ❸ Consider security-sensitive operations as sinks, and conduct source-to-sink analysis to detect vulnerabilities.

However, three main challenges arise with this workflow. In this section, we first discuss the challenges encountered (in §3.1) and then introduce our key ideas (in §3.2).

## 3.1. Challenges

**Challenge I: How to infer the user-accessible entry points by understanding the flexible semantics of the gateway configuration?** For traditional monolithic applications, existing works [44, 51, 61] consider all variables corresponding to HTTP request parameters (e.g., $_GET in PHP) as entry points (i.e., sources) to perform taint propagation analysis, and then report unprotected source-to-sink paths as potential vulnerabilities. The effectiveness of these works is based on the assumption that all entry points can be directly accessible by user requests. However, this does not hold in microservice applications due to the request routing control by gateway components, which means that many entry points are not exposed to users and can only be reached through inter-service communication. This leads to many false positives if existing taint analysis approaches are directly applied to microservice applications without thoroughly analyzing the request routing rules of the gateway (as shown in §5.4, our evaluation results demonstrate that the false positive rate reached as high as 60.14%).

Therefore, it is critical to understand the routing rules of the gateway configuration, as these rules dictate whether the entry points of a microservice are accessible to users or not. Nevertheless, this is not a trivial task. The primary challenge stems from the fact that routing rules are often configured with developer-customized and unstructured content, which complicates their interpretation. Besides, there are diverse gateway components (e.g., SpringCloud-Gateway, Zuul, APISIX), each with potentially hundreds of routing rules [27, 28]. Manually analyzing and modeling these rules is labor-intensive and impractical. Figure 4 illustrates an example of a gateway configuration file with four simple routing rules. The two rules on the left expose the `portal` and `api` services to users, while the two on the right block user requests, preventing the `user` and `admin` services from being accessible. This is primarily achieved through the `filter` field. The `AddHeader` and `PrefixPath` rules do not block requests, whereas the `SetResponseStatus` and `Denied` rules intercept user requests. It is clear that the configuration of such rules is highly flexible and can involve varying meanings and structures, making them difficult to model comprehensively.

```
// Allow access                    // Deny access
portal-route:                      user-route:
   path: /portal/**                   path: /user/**
   service: portal                    service: user
   filters: AddHeader=X-Portal        filter: SetResponseStatus=403

api-route:                         admin-route:
   path: /api/**                      path: /admin/**
   service: api                       service: admin
   filters: PrefixPath=/v1            filter: AddResp=X-Admin, Denied
```

Figure 4: An example of a gateway configuration.

**Challenge II: How to accurately establish connections between diverse inter-service communication senders and their corresponding receivers?** Microservice applications typically propagate data flow through inter-service communication. Taking Figure 3 as an example. In the `Portal` service, the `send()` method (line 4) places the user-provided `id` as a message into Kafka. Then, the `User` service retrieves this message through the `poll()` method (line 9). This communication mechanism allows data to propagate across different microservices. Clearly, this renders the existing approach of tracking such inter-service data flows in microservice applications, potentially missing many real vulnerabilities. As shown in §5.4, the false negative rate reached as high as 54.24%.

The primary challenge in analyzing inter-service communication arises from connecting the *senders* with their corresponding *receivers* when a communication operation is encountered, especially given the highly flexible implementation practices adopted in real-world systems. As shown in Figure 3, the `send` method (line 4) acts as the message sender, while the `poll` method (line 9) serves as the message receiver. To establish the connection between them, we first need to identify the sender from thousands of method calls and then accurately link it to the corresponding receiver, which may be located in another service. Given the diversity of communication components and the variability in implementation strategies, this task presents significant challenges.

**Challenge III: How to conduct effective taint analysis for detecting microservices vulnerabilities?** After determining the user-accessible entry points and understanding inter-service communication, an intuitive approach is to conduct taint analysis directly over the whole microservice application like a regular web application. However, this remains a challenging task.

In taint analysis of web applications, calling-context sensitivity is crucial for achieving high analytical precision. Context-insensitive approaches are highly efficient, analyzing a method once and applying the results uniformly across all its callsites. While this approach significantly reduces analysis overhead, it disregards the unique context of each method call site, leading to more false positives in vulnerability detection. In contrast, context-sensitive approaches analyze each callsite of a method repeatedly to achieve higher analytical precision. Nevertheless, this repetitive analysis introduces substantial overhead, increasing the risk of timeouts and out-of-memory issues that interrupt the analysis [31, 34]. In microservice applications, call chains can be extremely long due to the concatenation of multiple services, which exacerbates this issue. As demonstrated in §5.3 and §5.4, our evaluation reveals that the false negative rate due to out-of-memory exceptions reached as high as 50.85%, and the state-of-the-art approach, CodeQL [2], also produced four false negatives due to timeouts. Therefore, how to balance analytical precision and overhead in the security analysis of microservice applications presents a significant challenge.

## 3.2. Our Solution

To address these key challenges, we propose our novel solutions from three key perspectives: 1) identifying user-accessible entry points using an LLM-assisted approach, 2) establishing the connection for inter-service communication through a consistent communication mode and identifiers, and 3) effectively detecting vulnerabilities using a distance-guided strategy.

**Solution I: LLM-assisted Entry Points Identification.** Although the gateway configuration files are highly flexible and unstructured, making comprehensive rule modeling challenging, we observe that these routing rules frequently convey rich semantics, which serve as indicators of their intended behavior. Recent research has demonstrated that Large Language Models (LLMs) exhibit strong capabilities in understanding the semantics of natural language and code. This is because LLMs are trained on extensive datasets that include a substantial amount of both natural language and code, enabling them to effectively comprehend and analyze gateway configurations.

Therefore, we propose an LLM-assisted static analysis technique to analyze the gateway configuration file in order to identify user entry points. First, we apply a few-shot learning approach [35] to construct prompts and utilize the LLM to interpret the semantics of gateway configuration files, allowing us to extract the routing rules that forward requests to internal microservices. Then, we leverage static analysis techniques to extract all potential entry points from the application code and apply the extracted routing rules to map them accordingly. Only those entry points that are successfully mapped are considered exposed to users. These user-accessible entry points serve as the starting points for our analysis, offering a comprehensive understanding of how user requests interact with the microservices application. Take Figure 4 as an example. By leveraging the LLM-assisted approach, we can understand the semantics of the `SetResponseStatus=403` and `Denied` fields in the routing rules, thereby accurately identifying the entry points `/portal/**` and `/api/**` as user-accessible.

**Solution II: SDG-based Inter-service Communication Analysis.** Inter-service communication is fundamentally a network-based interaction mechanism that must adhere to specific protocols and standards. Based on the OWASP Microservice Cheatsheet [7], we observed two key principles required for successful communication between senders and the corresponding receivers: (1) consistent *mode*, i.e., the specific network request components, and (2) consistent *identifiers*, i.e., the symbols used to ensure correct communication addresses and communication channel consistency.

Based on these two key principles, we identify the *mode* and *identifiers* of senders and receivers, thereby establishing accurate connections. Specifically, for the *mode*, we followed guidelines from the Cloud Native Computing Foundation (CNCF) [1, 26] to identify the APIs of commonly used inter-service communication components (e.g., the `send` method in Figure 3). This allowed us to systematically determine the communication mode (detailed in §4.2.2). For the *identifier*, we extracted values of the arguments passed to the communication APIs, treating them as the identifiers linking the sender and receiver. However, the implementation of identifiers is highly flexible and difficult to model. For instance, in Figure 3, the identifier “user/query” is constructed using the `String.format` function and `op` variable (line 4). To address this, we employed backward data flow analysis to capture data flows associated with the identifier, replacing its variables with the values found at the terminal points (e.g., constant value “query”) of these data flows, thereby determining its final value.

To facilitate subsequent analysis, we further represented the connections between senders and receivers as edges within the Service Dependence Graph (SDG). Leveraging this, we can conduct a comprehensive analysis of taint propagation across microservices, thereby significantly enhancing our ability to detect inter-service vulnerabilities.

**Solution III: Distance-guided Context-sensitive Analysis.** In essence, the calling-context sensitive problem is a trade-off between analysis precision and efficiency. Therefore, a selective approach has become mainstream ([52, 60]), where context-sensitive analysis is applied only to callsites related to the analysis target to ensure precision, while context-insensitive analysis is used for unrelated callsites to improve efficiency. However, the fundamental challenge here is determining which callsites are relevant to the analysis target. To address this challenge, existing work often pre-defines patterns (e.g., $k$-limit [60] or code pattern [52]) to select interesting callsites for context-sensitive analysis. However, we have found that these patterns may only be applicable to specific scenarios and are not universally applicable (e.g., microservice applications).

Therefore, we have innovatively proposed a distance-guided strategy to conduct selective context-sensitive taint analysis. This strategy originates from our observation that call sites closer to the source-to-sink path typically have a greater impact on the accuracy of vulnerability detection, necessitating more precise analysis. Consequently, we dynamically select the level of context sensitivity based on the proximity of the analyzed callsite to the source-to-sink path within the call graph. Specifically, for callsites located directly on the source-to-sink path, our approach employs fully context-sensitive taint analysis. For callsites

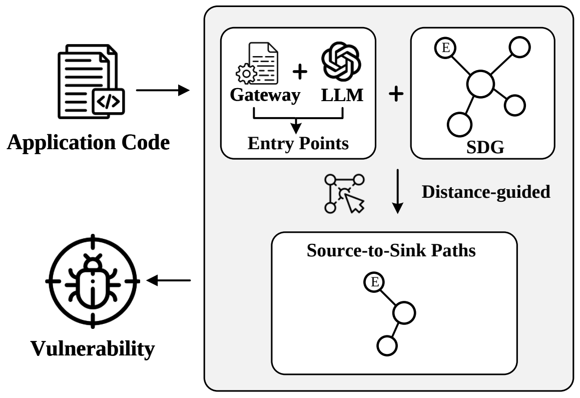


Figure 5: The Architecture of `MScan`.

that are further from this path, our approach calculates the shortest distance to the nodes on the source-to-sink path and progressively reduces the level of context sensitivity as this distance increases.

## 4. Methodology of `MScan`

In this section, we provide the design details of our approach, called `MScan`. Figure 5 illustrates the architecture of `MScan`, which consists of three key modules:

- *User-accessible Entry Point Identification (§4.1)*. This module focuses on understanding the semantics of the gateway configuration to accurately identify user-accessible entry points.
- *Service Dependence Graph Construction (§4.2)*. This module constructs the SDG to represent inter-service communication, thereby facilitating the analysis of taint propagation across microservices.
- *Vulnerability Detection (§4.3)*. This module leverages the inter-service and selective context-sensitive taint analysis to detect vulnerabilities within microservice applications.

### 4.1. User-accessible Entry Point Identification

In this module, `MScan` first utilizes an LLM-assisted method to understand the gateway configurations and extract the routing rules that forward requests to internal microservices. Next, `MScan` extracts all entry points from the application code and maps them to the extracted routing rules to further identify the exposed ones.

**Step I: LLM-assisted Routing Rule Extraction.** First, `MScan` utilizes a few-shot learning approach [35] to construct the prompts. As depicted by Figure 11 in Appendix §B, a typical few-shot prompt is structured into three main components: (1) the task description, (2) a set of K example queries with answers (known as K shots), and (3) the actual query that needs to be answered. `MScan` constructs its prompts as follows:

- ***Task description*** component offers a comprehensive explanation of the task at hand. This includes specifying the type of information that will be provided, outlining how the input should be processed, and detailing the expected format and content of the responses. Following best practices in prompt engineering [15], we specified a persona for the LLM as a *gateway configuration analyzer* within the prompt and defined the response format, i.e., presenting the routing rules in JSON format.
- ***K shots*** serve as standard models that enable the LLM to generate responses in the same format as the example answers and learn to perform tasks by identifying common patterns in routing rules within gateway configuration files, all without requiring modifications to the model's parameters. The format of the example queries is consistent with that of the actual query, with each query followed by its corresponding answer.
- ***Actual query*** provides contextual information about the task, allowing the LLM to execute the instructions effectively. `MScan` inputs the gateway configuration file as the actual query to the LLM. This configuration file typically has a fixed path or filename, making it easy to locate. Based on the LLM's response, `MScan` can accurately extract the routing rules that forward requests to internal microservices.

**Step II: User-accessible Entry Point Identification.** Then, `MScan` utilizes the extracted routing rules to identify exposed entry points. Specifically, `MScan` utilizes static analysis to extract the URIs of all entry points within the microservice. For each extracted URI, `MScan` uses a regular expression engine to evaluate whether any of the user-accessible routing rules match the URI. Entry points that successfully match these routing rules are designated as exposed. Given that routing rules of every gateway component support regular expression syntax, our approach, though seemingly straightforward, has demonstrated high effectiveness and generality. As illustrated in Figure 4, the `portal-route` routing rule defines that all URIs starting with `/portal` can be forwarded to the `Portal` service. However, the `filter` element `SetResponseStatus=403` of `user-route` routing rule determines that this rule will not be used to identify exposed entry points.

### 4.2. Service Dependence Graph Construction

In this module, `MScan` constructs the Service Dependence Graph (SDG) to represent the inter-service communication relationships within microservice applications.

**4.2.1. SDG Definition.** First, we utilize graph notation to rigorously define the formal representation of the SDG.

**Definition 1** (SDG)**.** The SDG is constructed on top of the Inter-procedural Control Flow Graph (ICFG) and represents inter-service communication in microservices through specialized nodes and edges. We define them below.

**Definition 2** (SDG Node)**.** The set of SDG nodes $N$ includes all nodes from the ICFG along with a new type of node, termed *I-Node*. Specifically, an *I-Node* uniquely represents each communication instance, connecting the sender to its corresponding receiver within an inter-service communication channel. To this end, each *I-Node* possesses the following properties:

- ***identifier***: The unique identifier for inter-service communication, typically a unique symbol (e.g., string), ensures correct message routing and communication consistency.
- ***mode***: The communication mode defines the APIs for message transmission and reception, influencing the interaction patterns between services.

We use $N_{icfg}$ to represent the set of nodes in the ICFG. Thus, the formal definition of $N$ is as follows:

$$N = I\text{-}Node \cup N_{icfg}$$

**Definition 3** (SDG Edge)**.** Next, we describe the composition of the SDG edge set $E$. Specifically, $E$ includes all edges from the ICFG, encompassing both intra- and inter-procedure control flow edges. Furthermore, to facilitate inter-service taint analysis, we enrich the SDG with *Data Flow Edges* and *Communication Edges*.

❶ **Intra-service Data Flow Edge** delineates the data dependencies between variables within a single microservice. Specifically, a data flow edge is a directed edge originating from a parameter or variable and terminating at another variable that depends on it. Through these data flow edges, we can accurately trace taint propagation within a single microservice, thereby establishing a robust foundation for inter-service taint analysis. We denote the set of data flow edges in the SDG as $D$.

❷ **Inter-service Communication Edge** represents message transfers between different microservices. It connects the sender and receiver with the same communication instance through an *I-Node*, thus characterizing inter-service communication and enabling taint analysis across microservices. The definitions of sender and receiver are as follows:

- *sender*: The ICFG node that produces and sends the message with an identifier to another microservice, typically implemented by a method call (denoted as $N_s$).
- *receiver*: The ICFG node that receives and consumes the message. The receiver retrieves the corresponding message from the sender based on the predefined identifier and communication mode (denoted as $N_r$).

We use $C$ to denote the set of communication edges in the SDG. The formal definition of $C$ is as follows, where $n_s$ is the sender ICFG node, $i$ is the identifier node, and $n_r$ is the receiver ICFG node:

$$C = \{(n_s, n_r) \mid \exists i \in I\text{-}Node,\ n_s \in N_s(i) \wedge n_r \in N_r(i)\}$$

We use $E_{icfg}$ to represent the set of edges in the ICFG. Therefore, the formal definition of $E$ is as follows:

$$E = D \cup C \cup E_{icfg}$$

**4.2.2. SDG Construction.** Then, we present how `MScan` constructs the Service Dependence Graph (SDG) based on the Inter-procedural Control Flow Graph (ICFG).

**I-Node Construction.** `MScan` employs a two-step approach to construct *I-Nodes* within a microservice application.

First, `MScan` identifies the inter-service communication mode, referred to as the *mode* property of the *I-Node* which defines the APIs for message transmission and reception. Specifically, `MScan` models the commonly used APIs of communication components based on the specifications from the Cloud Native Computing Foundation (CNCF) [1]. These APIs are broadly categorized into two types: synchronous (e.g., gRPC [5]) and asynchronous (e.g., Kafka [6] message queue). `MScan` detects invocations of these modeled APIs within the microservice application to determine the communication mode. We detailed the specific communication modes supported by `MScan` in Appendix §C. Additionally, the sink list in `MScan` is highly customizable. We acknowledge that there may be some communication components not covered, but enabling `MScan` to cover them necessitates only minimal manual effort.

Second, `MScan` extracts the arguments of these APIs that serve as identifiers and assigns them as the *identifier* property of the *I-Node*. Specifically, `MScan` performs backward data flow analysis for each argument of these APIs. This analysis traces the data flow from the point of the argument back through the program to its origin, effectively capturing the entire context in which the argument is used. Once the program data flows are obtained, `MScan` replaces the variables in the identifier with the values found at the terminal points of these data flows, typically constant values. This process involves examining the final values assigned to the variables and using these values to define the identifiers accurately.

**Inter-service Communication Edge Connection.** `MScan` utilizes the constructed *I-Nodes* to establish inter-service communication edges between senders and their corresponding receivers, thereby constructing the comprehensive SDG. Specifically, `MScan` leverages the modeled communication APIs to identify the sender and receiver nodes within the ICFG. This involves pinpointing the exact locations in the code where the communication APIs are invoked, thus identifying the nodes that act as senders and receivers in the communication process. Subsequently, `MScan` connects communication edges between the corresponding *I-Node* of senders and receivers that share the same *identifier* and *mode*. By linking these nodes, `MScan` effectively maps out the communication pathways, illustrating how messages traverse between different microservices.

**Intra-service Data Flow Edge Connection.** `MScan` aims to establish data flow edges within the SDG. Specifically, `MScan` applies the well-established IFDS (Inter-procedural, Finite, Distributive, Subset) algorithm [57] to construct these data flow edges. The IFDS is a widely used and highly effective method for solving a broad class of inter-procedural data flow analysis problems by reducing them to generalized graph reachability tasks. This approach has been extensively adopted in numerous prior works [31, 50]. By utilizing the IFDS algorithm, `MScan` constructs data flow edges between variables exhibiting data dependencies, thereby facilitating the accurate tracking of taint propagation within single microservices.

**Construction Example.** We refer to Figure 6 to further

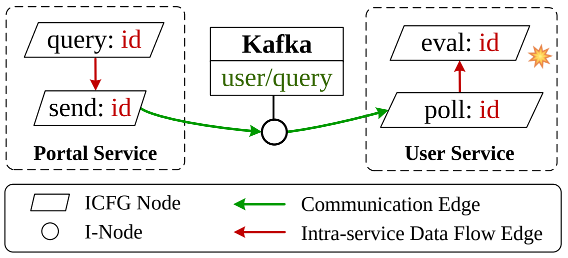


Figure 6: SDG of the example code in Figure 3.

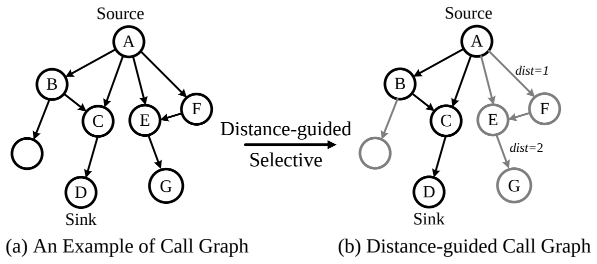


Figure 7: An example of Distance-guided Strategy.

explain how MScan constructs the SDG for the code in Figure 3 and demonstrate how MScan utilizes the SDG to perform inter-service data flow analysis.

Firstly, MScan constructs the *I-Nodes* for both the sender and receiver. For the sender, MScan first identifies the *mode* property as Kafka based on the APIs listed in §C. Then, using backward dataflow analysis, it determines that the *identifier* property is "user/query" (line 4). Accordingly, MScan constructs the $I\text{-}Node_{(\texttt{user/query,Kafka})}$ for the `send` method (line 5). For the receiver, MScan applies the same approach to construct its *I-Node* for the `poll` method (line 9). Secondly, MScan establishes the communication edge between the `send` method in the `Portal` service and the `poll` method in the `User` service due to their shared *I-Node*, as depicted by the green line in Figure 6. Finally, by leveraging the intra-service data flow edge (shown by the red lines in Figure 6), MScan successfully connects the inter-service data flow and traces the taint flow from the `id` parameter of the `query` method (line 2) to the `id` parameter of the `eval` method (line 10). By utilizing the SDG, MScan performs effective inter-service data flow analysis.

## 4.3. Vulnerability Detection

In this section, we elaborate on the details of our advanced distance-guided strategy and its efficacy in detecting microservice vulnerabilities.

**4.3.1. Selective Context-Sensitive Taint Analysis.** We first introduce the concept of context sensitivity in taint analysis, and then explain how our distance-guided strategy works for conducting selective context-sensitive taint analysis, followed by an example to demonstrate its effectiveness in microservice security analysis.

**Context Sensitivity in Taint Analysis.** Context sensitivity is a fundamental mechanism for achieving high analysis precision, as it distinguishes each method callsite based on its calling context, thereby allowing each callsites to be analyzed uniquely according to their respective contexts [47, 58]. Take Figure 7 (a) as an example. Suppose the context sensitivity is set to 3. For the method callsite `D`, it records two context objects: $[A, C, D]$ and $[B, C, D]$. However, when the context sensitivity is reduced to 2, it only records one context object: $[C, D]$. These objects are stored and analyzed within memory. Clearly, the lower the context sensitivity, the less precise the taint analysis becomes. Conversely, as context sensitivity increases, the memory and time required for recording and analysis also grow significantly.

**Distance-guided Strategy.** The distance-guided strategy adjusts the degree of context sensitivity based on the proximity of the analyzed method to the source-to-sink path, thereby focusing more effort and resources on security-critical analyses. We use formalized equations to describe the distance-guided strategy as follows. Specifically, we refer to the method nodes on the source-to-sink path as dominant nodes, denoted as $DN$, and other method nodes as un-dominant nodes, denoted as $UN$. We define $dist(n, DN)$ as the distance between a node $n$ and its nearest dominant node in the call graph. For each node $n$, its context-sensitivity $S(n)$ is defined as follows. $S_\text{m}$ represents the maximum context sensitivity, which is equal to the length of the longest source-to-sink path. $K$ represents the factor to adjust the influence of distance (with a default value of 2).

$$S(n) = \begin{cases} S_\text{m} & n \in DN \\ \max\left(1, \frac{S_\text{m}}{\text{dist}(n, DN)^2 \cdot K}\right) & n \in UN \end{cases}$$

Following this, for methods directly within the source-to-sink path, MScan employs fully context-sensitive taint analysis, i.e., $S_\text{m}$. For other methods that belong to the set of un-dominant nodes, MScan calculates the shortest distance to the nodes of the source-to-sink path, i.e., $dist(n, DN)$, and correspondingly reduces context sensitivity as this distance increases.

Figure 7 (b) illustrates an example of the distance-guided strategy. The longest source-to-sink path is 4, meaning that $S_\text{m}$ is 4. For method callsites on the source-to-sink path, i.e., $A$, $B$, $C$, and $D$, MScan treats them as dominant nodes and performs the context-sensitive analysis. As a result, MScan maintains a total of 8 context objects between these callsites: $[A, B]$, $[A, C]$, $[A, B, C]$, $[A, B, C, D]$, $[A, C, D]$, $[B, C]$, $[B, C, D]$, $[C, D]$. For other method calls, MScan considers them as non-dominant nodes and reduces context sensitivity as the distance from the source-to-sink path increases. For instance, the callsite $G$ is at a distance of 2 from the source-to-sink path, so its context sensitivity is reduced from $S_\text{m}$ (i.e., 4) to 1, which decreases the number of maintained context objects from 4 to 1, i.e., $[E, G]$. As the complexity of the call graph increases, the advantages of our distance-guided strategy become increasingly pronounced (as detailed in §5.4).

**4.3.2. Microservice Vulnerability Detection.** MScan utilizes the constructed SDG to perform inter-service context-sensitive taint analysis for detecting vulnerabilities in microservice applications. The process can be divided into two primary steps. Initially, MScan treats user inputs (i.e., parameters of user-accessible entry points) as the taint *sources*, and employs the data flow edges of the SDG to trace the propagation path of the taint. This step is highly effective for intra-service taint analysis, accurately identifying variables that have data flow dependencies with the taint. Then, when inter-service communication is encountered, MScan uses the SDG's communication edges to locate the receiver in other microservices, thereby continuously tracking the propagation path of the taint across different microservices. Finally, if the tainted variable could be used in a security-sensitive operation, i.e., the *sink*, MScan will report a vulnerability.

## 5. Evaluation

### 5.1. Experimental Setup

**Implementation.** We have developed a prototype of MScan targeting Java-based microservice applications. Specifically, our prototype is built on Tai-e [60], a state-of-the-art static analysis framework, and consists of over 7,000 lines of Java code. We employ the standard static analysis to construct the inter-procedural control flow graph (ICFG) and extend Tai-e's taint analysis plugin to build the service dependence graph (SDG), thereby performing inter-service and context-sensitive taint analysis.

For the security-sensitive sinks, we referred to industrial static analysis tools and existing works [44, 51, 61] to model the method signatures of various sensitive sinks. For example, the rule `<groovy.lang.GroovyShell: java.lang.Object.evaluate(String),0>` models the sinks for groovy-code injection. To date, MScan supports 8 vulnerability types, i.e., command injection, SSRF, XXE, SSTI, Groovy-code injection, SpEL injection, SQL injection, and arbitrary file operations. Extending MScan to support additional types requires minimal effort.

For the sanitizer, MScan adopted models of common sanitizers from CodeQL, e.g., `PathSanitizer` [29], to stop the taint propagation. These sanitizers effectively minimize MScan's false positives.

For the LLM, MScan leverages *gpt-4o* [4] to perform the gateway semantic understanding task and we consumed about 47k tokens during the evaluation.

**Experiments.** All experiments were conducted on a server equipped with a 64-core Intel Xeon Gold 6242 CPU and 256 GB of memory, running Ubuntu 20.04. Our evaluation seeks to answer the following four research questions:

- RQ1: How effective is MScan in detecting taint-style vulnerabilities within real-world microservice applications?
- RQ2: How effective is MScan compared with the state-of-the-art techniques?
- RQ3: How do the different components of MScan contribute to its success?
- RQ4: How efficient is MScan in performing the analysis?

TABLE 1: Breakdown of our evaluation dataset.

| Open-source Applications | # Stars | # LoCs |
|---|---|---|
| Apollo | 29,170 | 44,815 |
| Yudao-cloud | 16,309 | 134,585 |
| Piggymetrics | 13,258 | 3,286 |
| Mall-swarm | 11,417 | 65,933 |
| Paascloud-master | 9,796 | 25,149 |
| basemall | 8,966 | 42,571 |
| SpringBlade | 6,397 | 6,337 |
| Spring-Cloud-Platform | 6,339 | 7,554 |
| Mall4cloud | 5,732 | 27,272 |
| Pig | 5,604 | 13,071 |
| Lamp-cloud | 5,369 | 33,478 |
| Microservices-platform | 4,485 | 12,508 |
| RuoYi-Cloud-Plus | 4,303 | 41,519 |
| PassJava-Platform | 2,564 | 10,075 |
| Spring-boot-cloud | 2,121 | 5,984 |
| Youlai-mall | 2,027 | 14,065 |
| Open-mall | 1,985 | 30,540 |
| Gulimall-learning | 1,994 | 20,714 |
| SuperMarket | 1,947 | 2,821 |
| Mogu_blog_v2 | 1,572 | 26,136 |
| Light-reading-cloud | 1,277 | 3,613 |
| Novel-cloud | 1,170 | 6,154 |
| RuoYi-Cloud | 1,152 | 18,245 |
| Spring-cloud-dataflow | 1,113 | 112,630 |
| Sitewhere | 1,020 | 38,784 |
| **Industrial Applications** | **# Stars** | **# LoCs** |
| T*** | / | 84,978 |
| M*** | / | 58,387 |
| M*** | / | 35,520 |
| B*** | / | 173,155 |
| Y*** | / | 13,280 |

**Dataset.** In all, our dataset includes 25 open-source and 5 industrial microservice-structured web applications. We provide detailed information about these applications in Table 1. The construction process is as follows:

- **Open-source Applications.** We collected microservice applications from popular open-source repositories (e.g., GitHub [3]) based on the following criteria. 1) Each application has over 1,000 stars in its respective repositories, confirming its popularity. In addition, we exclude tutorial applications, such as demos and study projects. 2) Each

TABLE 2: Distribution of detected vulnerabilities (RQ1).

| Vulnerability Type | TP | FP | Prec(%) |
|---|---|---|---|
| Intra-service | 27 | 12 | 69.23% |
| Inter-service | 32 | 11 | 74.42% |
| **Total** | 59 | 23 | 71.95% |

application is required to possess a microservice-oriented architecture, and we determined this by checking for the presence of the keyword 'microservice' in the project description and analyzing the frameworks employed in the project (followed by [55]). As a result, we collected 25 open-source microservice applications, most of which have also been utilized in existing research [37, 48, 55].

- **Industrial Applications.** We collaborated with a world-leading fintech company that provides services to billions of users. They generously supplied us with 5 industrial applications, which we have also included in our evaluation dataset. Note that the company has a well-established SDL (Secure Development Lifecycle) team. Each application deployed online undergoes multiple rounds of manual inspection, including comprehensive code audits and extensive black-box testing. Therefore, identifying unknown vulnerabilities within these applications is really challenging.

## 5.2. RQ1: Effectiveness

In this part, we evaluated the effectiveness of `MScan` in detecting taint-style vulnerabilities in microservice applications across the dataset.

**Result Overview.** Overall, `MScan` reported a total of 82 potential vulnerabilities. Table 2 presents the details. Specifically, 39 of these reported vulnerabilities are of the intra-service type, while 43 are of the inter-service type. Through manual inspection of each case, we ultimately confirmed 59 taint-style vulnerabilities, including SQL injection (SQLi), XML external entity injection (XXE), server-side request forgery (SSRF), groovy code injection (GCi), and arbitrary file operations (AFW & AFR). These vulnerabilities affected 11 open-source and 2 industrial microservice applications.

**Bug Disclosure.** These detected vulnerabilities pose significant security threats to the target applications. For the 39 vulnerabilities detected in open-source applications, attackers can exploit these vulnerabilities to steal data stored in databases, upload malicious files to the application, and even control the entire server. Consequently, we promptly notified the developers of all confirmed vulnerabilities in the affected applications. Some of these vulnerabilities were immediately addressed and patched, while for others, we are still actively communicating with the developers to assist them in understanding and resolving the issues. As of now, as shown in Table 5 in §A, we have received 31 CVE identifiers in acknowledgment, including CVE-2024-22263 [20] in the Spring Projects [18], which has a CVSS score of 8.8. For the 20 vulnerabilities detected in industrial applications, all were previously unknown and newly identified. These vulnerabilities can lead to arbitrary code execution and even take over the entire web server, severely compromising the integrity of user data and the security of company systems. These findings demonstrate `MScan`'s practical utility.

**False Positives.** We comprehensively analyzed all the 23 false positives, and their causes can be mainly divided into three aspects. Firstly, 12 false positives were caused by *developer-customized sanitizers*. To prevent malicious user input from flowing into the parameters of security-sensitive operations, developers implement checks on the data flow or control flow of user inputs, i.e., sanitizer. However, these developer-customized sanitizers are highly flexible, making them difficult to identify through simple modeling, causing `MScan` to report protected paths as potential vulnerabilities, leading to false positives. Secondly, 9 false positives were caused by *unreachable vulnerable code*. Since `MScan` performs static analysis to trace paths from source to sink without constraint solving for the path, it reports vulnerable paths that are actually unreachable due to unsatisfied constraints, resulting in false positives. Thirdly, 2 false positives were caused by *the inability to distinguish the developer's design intent*. For example, in yudao-cloud [30], developers provided application administrators with the capability to execute arbitrary SQL statements for efficient management of application data. It was apparent that there is a data flow from user input to database operations, prompting `MScan` to report it as a vulnerability. After active discussions with the developers, it was determined that this functionality was an intended feature. Consequently, we reclassified this instance as a false positive by `MScan`.

## 5.3. RQ2: Comparison

In this part, we compare the effectiveness of `MScan` with the state-of-the-art technique (i.e., `CodeQL` [2]) across the entire dataset.

**Baseline Setup.** `CodeQL` is the current state-of-the-art technique widely used in static vulnerability detection. `CodeQL` comes with a wide range of built-in, ready-to-use rules, enabling it to detect various types of CWE in web applications [23]. To ensure a fair and comprehensive comparison, three authors of the paper (each with five years of security research experience) manually reviewed CodeQL's rules and documentation for 65 built-in CWEs [23, 24], ultimately selecting 19 taint-style vulnerability-related CWEs and applied them to analyze all the applications in our dataset.

**Ground Truth Construction.** Given that comparing the effectiveness of baseline with `MScan` requires labeling all vulnerabilities in the dataset, which is not feasible [39, 44]. Therefore, we employed a commonly used method [61] by constructing a ground truth from the combined set of vulnerabilities identified by both `MScan` and `CodeQL` in our dataset. All vulnerabilities in the ground truth were meticulously verified and tested with PoC, ensuring that only actual vulnerabilities were included. Ultimately, our ground truth consists of 59 verified vulnerabilities.

TABLE 3: Comparison between MScan and CodeQL (RQ2).

| Baselines | TP | FP | FN | Prec(%) | Recall(%) |
|---|---|---|---|---|---|
| CodeQL | 23 | 35 | 36 | 39.66% | 38.98% |
| MScan | 59 | 23 | 0 | 71.95% | 100.00% |

**Result Overview.** Table 3 provides a detailed comparison of the effectiveness of MScan and CodeQL across the entire dataset. Overall, MScan demonstrates better performance, surpassing CodeQL by 90.72% in precision and 156.54% in recall. More specifically, when tested against the ground truth of 59 vulnerabilities, MScan identifies all of them, whereas CodeQL detects only 23 vulnerabilities with 35 false positives. These results underscore the superior capability of MScan in effectively detecting taint-style vulnerabilities within microservice applications.

**False Positive Analysis.** In total, CodeQL reported 35 false positives. After a thorough analysis, we found that apart from the 14 cases also reported by MScan, CodeQL reported 21 additional false positives. The first reason is that *the gateway routing rules* prevent certain entry points from being accessed by users. As described in §3.1, developers may configure the gateway to restrict access to some internal microservices for security purposes. Consequently, the vulnerable endpoints within these microservices remain inaccessible, leading to false positives. The second reason is CodeQL's *limited context-sensitive approach*. Specifically, CodeQL employs a context-sensitive strategy similar to $k$-CFA [54]. As a result, for certain sinks deeply hidden within the system, CodeQL is unable to distinguish their calling contexts, which leads to false negatives during detection.

**False Negative Analysis.** For the 36 false negatives, CodeQL missed them due to two main reasons. On one hand, 32 false negatives were caused by *incomplete call graph construction*. As described in §3.1, CodeQL is unable to understand inter-service communication mechanisms, making it incapable of tracking inter-service data flow, rendering it incapable of tracking inter-service data flow and thus missing all inter-service vulnerabilities. On the other hand, 4 false negatives are attributed to the *context-sensitive analysis approach* of CodeQL. Specifically, while the $k$-CFA context-sensitive strategy of CodeQL is effective in some cases, for large applications with a vast number of source-to-sink paths, CodeQL encountered timeout errors, leading to missed vulnerabilities.

## 5.4. RQ3: Ablation Study

In this part, we conducted an ablation study to demonstrate the necessity of each key component of MScan to accurate and efficient vulnerability detection.

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

TABLE 4: Ablation study for three variants of MScan (RQ3).

| Baselines | TP | FP | FN | Prec(%) | Recall(%) |
|---|---|---|---|---|---|
| MScan-NoEntry | 59 | 89 | 0 | 39.86% | 100% |
| MScan-NoSDG | 27 | 12 | 32 | 69.23% | 45.76% |
| MScan-CS | 29 | 11 | 30 | 72.50% | 49.15% |
| MScan-CS-2call | 59 | 251 | 0 | 19.03% | 100.00% |
| MScan | 59 | 23 | 0 | 71.95% | 100.00% |

- *MScan-NoEntryDet.* In this variant, we disabled a key component of the MScan, namely *User-accessible Entry Point Identification*, leading to the variant directly treating all entry points, i.e., user-accessible and user-inaccessible entry points, as sources for taint analysis.
- *MScan-NoSDG.* In this variant, we disabled the *SDG Construction* component of the MScan, which means the variant relies solely on intra-service taint analysis capabilities to detect vulnerabilities.
- *MScan-CS.* In this variant, we disabled the *Distance-guided Strategy* of MScan and employed the no-selective context-sensitive strategy, i.e., fully context-sensitive, to assess its critical role in microservice vulnerability detection.
- *MScan-CS-2call.* In this variant, we also disabled the *Distance-guided Strategy* of MScan and employed a SOTA $k$-limit selective strategy from Tai-e [60]. Following the experimental setup described in Section 6.5 of Tai-e, we set $k$ to 2 (i.e., 2-call context-sensitive), which only tracks method call chains up to a depth of 2. For any callsites beyond this depth, the analysis degrades to a context-insensitive approach.

**Result Analysis.** Table 4 provides a breakdown of the comparison results between MScan and its three variants. It is clear that these four key components are essential for effective vulnerability detection in microservice applications. A detailed analysis of the results is as follows:

❶ *MScan vs. MScan-NoEntryDet.* In this phase, we detail MScan's identification of entries across the entire dataset and emphasize the importance of the exposed entry point determination component by comparing the vulnerability detection results of MScan and MScan-NoEntry. Overall, MScan identified 4611 entry points within the dataset, of which 1454 are unexposed, meaning they are not exposed to users. This indicates that over 31% of entry points are not directly accessible by users in these microservice applications. Incorrectly treating these unexposed entries as sources for taint analysis could result in false positives.

As shown in Table 4, MScan-NoEntry exhibited a decline in accuracy to 39.86% compared to the full version of MScan. We conducted a thorough analysis of the additional false positives produced by MScan-NoEntry and discovered that all these false positives stemmed from internal entries. Specifically, MScan-NoEntry misclassified unexposed internal entry points as sources, thereby resulting

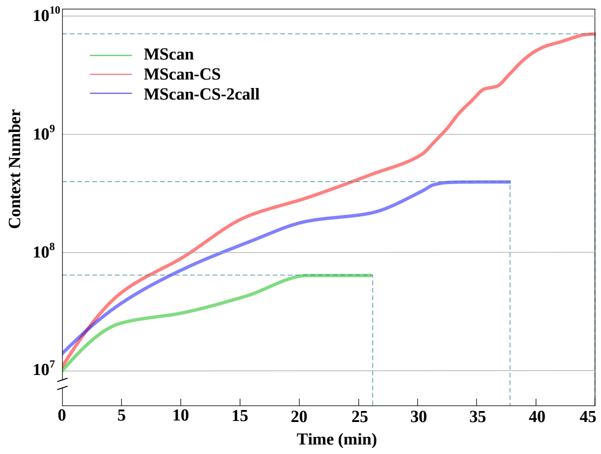


Figure 8: Analysis efficiency of `MScan` and `MScan-CS` in large-scale applications, i.e., yudao-cloud [30].

in a substantial number of false positives.

❷ *MScan vs. MScan-NoSDG.* In this phase, we demonstrate the accuracy of constructing the SDG's critical communication edges and compare the vulnerability detection results between `MScan` and `MScan-NoSDG`. Overall, `MScan` identified 1,050 instances of inter-service communication across 11 different communication mechanisms (as detailed in Appendix §C) in the entire dataset and successfully determined the target addresses for all these communications, thereby establishing the corresponding communication edges in the SDG. We manually verified all these edges and found that 16 instances could not determine communication addresses through dataflow analysis (e.g., addresses containing SpEL variables), which hinders accurate SDG construction. Nevertheless, through detailed manual analysis, we discovered that these instances did not result in any false negatives in vulnerability detection. This high-quality SDG thus provides a solid foundation for effective inter-service vulnerability detection.

For the `MScan-NoSDG` variant, since it lacks the capability to construct an SDG, it is unable to track taint propagation across microservices. Consequently, it failed to identify any vulnerabilities in the microservice applications within our dataset. as shown in Table 4, the `MScan-NoSDG` variant failed to detect any inter-service vulnerabilities and its accuracy dropped to 45.76%. This clearly highlights the critical role of the SDG component in the vulnerability detection process.

❸ *MScan vs. MScan-CS(-2call).* To clearly demonstrate the effectiveness of `MScan`'s distance-guided strategy, we conducted a comparative analysis of the number of context objects maintained by `MScan`, `MScan-CS`, and `MScan-CS-2call` during vulnerability detection. Using the largest application in our open-source dataset, yudao-cloud [30], which has over 130k lines of code, we observed significant differences. As shown in Figure 8, `MScan-CS`'s number of contexts continuously increased, reaching approximately $10^{10}$, and eventually caused an out-of-memory exception after about 45 minutes. In contrast, `MScan`'s distance-guided strategy dynamically adjusted the number of maintained contexts, reducing them by at least two orders of magnitude, from $10^{10}$ to $10^{8}$ (note that we cannot evaluate the exact order of magnitude due to the early interruption of the `MScan-CS`).

We further present the results of the two variants across the entire dataset to underscore the importance of the distance-guided strategy. As shown in Table 4, compared to `MScan`, the recall rate of `MScan-CS` is merely 49.15%, indicating that it missed a significant portion of the vulnerabilities detected by `MScan`. This is because `MScan-CS` could only complete the analysis for 11 applications in the entire dataset. Within the experimental setup outlined in §5.1, it encountered out-of-memory exceptions when analyzing other applications, resulting in a high number of false negatives. For the `MScan-CS-2call` variant, it successfully completed all analyses. However, since the 2-call context-sensitive strategy was insufficient to analyze the extensive call chain of microservice applications, it degraded into a context-insensitive analysis, leading to many false positives, with a precision rate of only 19.03%.

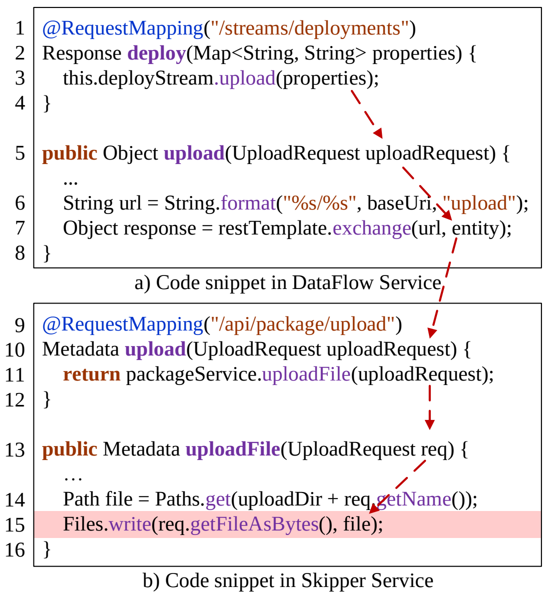
## 5.5. RQ4: Efficiency

In this experiment, we evaluated the performance of `MScan` in conducting end-to-end analysis across the entire dataset. The end-to-end analysis time for `MScan` encompasses both the SDG construction and the inter-service taint analysis phases. Overall, `MScan` analyzed the 30 applications in the dataset in a total of 8.45 hours, averaging 16.9 minutes per application. Considering the complexity inherent in microservice calling contexts, we believe that the analysis time remains within acceptable and manageable limits. The efficiency of `MScan` can be attributed to its effective distance-guided strategy, which significantly reduces the overhead associated with analyzing sink-unrelated method calls, thereby accelerating the analysis process.

## 6. Case Study

In this section, we showcase some interesting taint-style vulnerabilities of inter-service type detected by `MScan` in highly popular applications, further illustrating the high risk posed by these vulnerabilities and demonstrating the practical utility of `MScan` in real-world scenarios.

**Case I:** *AFW vulnerability in Spring-Cloud-Dataflow (Inter-service Communication via RestTemplate [16]).* The spring-cloud-dataflow [19] is a microservices-based toolkit developed under the Spring Project [18] for building streaming and batch data processing pipelines, with over 1,100 stars on GitHub. As depicted in Figure 9, `MScan` identified an arbitrary file write (AFW) vulnerability within the application, enabling attackers to gain control over the application server by injecting malicious files. The identified vulnerability involves the interplay between the `DataFlow` and `Skipper` microservices within the application. In the



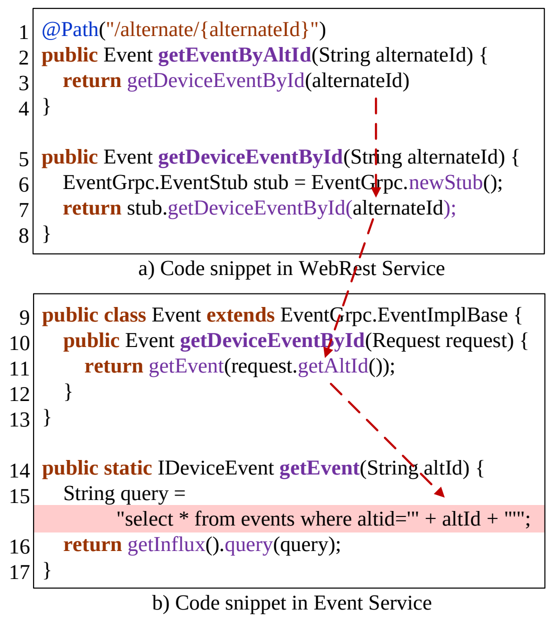
```
 1  @RequestMapping("/streams/deployments")
 2  Response deploy(Map<String, String> properties) {
 3     this.deployStream.upload(properties);
 4  }

 5  public Object upload(UploadRequest uploadRequest) {
       ...
 6     String url = String.format("%s/%s", baseUri, "upload");
 7     Object response = restTemplate.exchange(url, entity);
 8  }
```

```
 9  @RequestMapping("/api/package/upload")
10  Metadata upload(UploadRequest uploadRequest) {
11     return packageService.uploadFile(uploadRequest);
12  }

13  public Metadata uploadFile(UploadRequest req) {
       …
14     Path file = Paths.get(uploadDir + req.getName());
15     Files.write(req.getFileAsBytes(), file);
16  }
```

Figure 9: Arbitrary File Write vulnerability in Spring-Cloud-Dataflow application (Spring Project on GitHub).



```
 1  @Path("/alternate/{alternateId}")
 2  public Event getEventByAltId(String alternateId) {
 3     return getDeviceEventById(alternateId)
 4  }

 5  public Event getDeviceEventById(String alternateId) {
 6     EventGrpc.EventStub stub = EventGrpc.newStub();
 7     return stub.getDeviceEventById(alternateId);
 8  }
```

```
 9  public class Event extends EventGrpc.EventImplBase {
10     public Event getDeviceEventById(Request request) {
11        return getEvent(request.getAltId());
12     }
13  }

14  public static IDeviceEvent getEvent(String altId) {
15     String query =
             "select * from events where altid='" + altId + "'";
16     return getInflux().query(query);
17  }
```

Figure 10: SQL injection vulnerability in Sitewhere application (over 1k stars on Github).

`DataFlow` service, the `deploy` method acts as a user-accessible entry point, allowing users to deploy tasks via HTTP requests to the `/streams/deployments` endpoint. Subsequently, user input flows into the `upload` method (line 3) and communicates with the `Skipper` microservice through the `restTemplate.exchange` method (line 7). Through this inter-service communication, the user-provided `filename` is directly passed to the `write` method within the `Skipper` service (line 15), enabling attackers to manipulate the file storage path on the server, thus leading to an arbitrary file write vulnerability. We reported this critical issue to the Spring Project and received a CVE identifier (CVE-2024-22263) with a CVSS score of 8.8.

**Case II:** *SQL injection vulnerability in Sitewhere (Inter-service Communication via gRPC [5]).* The sitewhere [17] is a highly popular (over 1.1k stars on Github) industrial strength open-source application enablement platform, which has been utilized in existing research [48]. Figure 10 illustrates a SQL injection vulnerability reported by `MScan` in this application. The vulnerability involves the `WebRest` and `Event` microservices. In the `WebRest` microservice, user input starts from the user-accessible entry point (`getEventByAltId` in line 3) and flows into the `getDeviceEventById` method (line 7), which communicates with the `Event` microservice via the gRPC framework (line 10). Subsequently, in the Event microservice, the user input is passed into the `getEvent` method and is ultimately concatenated into an SQL query (line 15), thereby leading to a SQL injection vulnerability. Given the severity of this vulnerability, we promptly reported it to the developers of the affected application and engaged in active discussions to devise a solution. Consequently, we were granted a CVE identifier (CVE-2024-37827).

## 7. Discussion

**Legality and Ethicality.** This study has not presented any legal or ethical issues. We obtained the source code for local analysis and responsibly reported all detected vulnerabilities in open-source applications to the CVE Numbering Authority (CNA), and also assisted companies in fixing vulnerabilities in industrial applications. Additionally, we have contacted all the developers regarding the vulnerabilities found in §5.2, and will continue to communicate with them throughout the vulnerability disclosure process.

**Future Work.** Like other static analysis approaches, the prototype of `MScan` also faces analysis difficulties. On one hand, `MScan` may produce false negatives due to dataflow interruptions during the analysis process. This can be mitigated by adding transfer rules to the taint analysis process [43]. On the other hand, `MScan` may report false positives due to the complex and diverse sanitizers. Identifying sanitizers is an inherent challenge in static taint analysis, as noted in many existing works [32, 51].

## 8. Related Work

**Taint-style Vulnerability Detection.** In recent years, the techniques for automatically detecting taint-style vulnerabilities have been extensively studied.

A commonly used technique is static taint analysis. Existing works in this area can be broadly categorized into two lines. The first line of work adopts traditional taint analysis methods [32, 45, 51], which primarily focus on analyzing taint propagation within a single application. However, these approaches are not specifically designed for microservice applications and are therefore inadequate for addressing the challenges presented in this work. The second line of work attempts to perform cross-application static taint analysis in both web and binary domains [33, 38, 42, 46, 56]. Similar to the objective of MScan, these studies propose various techniques to establish data flow relationships across applications, thereby enabling the detection of cross-application taint-style vulnerabilities. However, applying these works directly to microservice-structured web applications does not yield effective results.

Specifically, DBTaint [38] and MiMoSA [33] observed that cross-application taint propagation may occur through database operations. Therefore, they modeled database operations and associated taint labels with stored values in the database to detect cross-application taint-style vulnerabilities in web applications. However, as shown in §5, microservice applications often contain inter-service taint-style vulnerabilities that do not involve database access, which DBTaint is unable to detect. CACG [46] analyzed the qualified names of cross-application communication classes and created edges between method calls and definitions with identical qualified names across different applications, thereby constructing a Cross-Application Call Graph targeting web applications. However, this call graph only supports coarse-grained method call analysis and cannot perform inter-service data flow analysis. Additionally, it ignores the gateway component, leading to a high number of false positives. Karonte [56] and Mango [42] analyzed hard-coded addresses (referred to as unique data keys by the authors) used in inter-process communication (IPC) to construct the data flow between the processes (or binaries) involved in the communication. This approach effectively helps detect cross-binary vulnerabilities. We appreciate their strategy of analyzing hard-coded communication addresses, which provides valuable insights for designing our service-dependence graph. However, in microservice applications, establishing inter-service communication links is more challenging (e.g., communication addresses may contain variables, making them difficult to extract). Therefore, directly applying their approaches to analyze microservice-based web applications is infeasible.

Another widely employed technique is dynamic testing (e.g., fuzzing) [39, 40, 44, 61]. Unfortunately, in the context of microservices, these approaches face three inherent limitations. First, these approaches require significant manual effort to deploy and coordinate multiple independent services. Second, existing fuzzing techniques struggle to gather key information (e.g., post-instrumentation basic block coverage) across independently deployed microservices. Third, testing throughput is reduced due to latency introduced by inter-service communication. These inherent limitations make their application to vulnerability detection in microservices less than ideal.

**Microsevice Security.** Recently, the security of microservice applications has garnered attention [48, 53, 59]. For instance, Minna *et al.* [53] performed a systematization of knowledge about the run-time security of microservices. ThunQ [59] highlighted that application-level access control policies play a crucial role in mitigating risks by preventing unauthorized access to the microservice application. Li *et al.* [48] attempted to mitigate microservices from being abused by other compromised microservices through the automatic generation of access control policies. However, taint-style vulnerabilities were not their primary focus and thereby were rarely explored or studied. Furthermore, their techniques cannot be directly applied to detect taint-style vulnerabilities in microservice applications.

**LLM-assisted code analysis.** Recent advancements in code analysis techniques using LLMs [41, 49, 62] have demonstrated that LLMs can effectively understand code semantics and perform various code analysis tasks. These developments inspire us to employ LLMs to analyze routing rules within gateway configuration files.

## 9. Conclusion

In this paper, we propose MScan, a novel security-vetting approach that can automatically detect injection-based vulnerabilities within microservice applications. Leveraging an inter-service taint analysis technique, MScan can effectively detect intra- and inter-service microservice vulnerabilities. We evaluated MScan on 30 real-world Java-based microservice applications. Overall, MScan discovered 59 high-risk 0-day vulnerabilities and 31 of them have been assigned with CVE identifiers. We hope our work can aid the community in addressing the rising threats of microservice 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, 62202106). 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). 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.

## References

[1] “CNCF cloudevents,” https://cloudevents.io/.
[2] “CodeQL,” https://codeql.github.com/.
[3] “Github,” https://github.com/.
[4] “GPT-4o,” https://openai.com/index/hello-gpt-4o/.
[5] “gRPC,” https://grpc.io/.
[6] “Kafka,” https://kafka.apache.org/.
[7] “Microservice CheatSheet in OWASP,” https://cheatsheetseries.owasp.org/cheatsheets/Microservices_based_Security_Arch_Doc_Cheat_Sheet.html.
[8] “Microservice in Amazon,” https://blog.dreamfactory.com/microservices-examples.

[9] "Microservice in Twitter," https://thenewstack.io/how-airbnb-and-twitter-cut-back-on-microservice-complexities/.

[10] "Microservice in Uber," https://www.uber.com/en-JP/blog/tech-stack-part-one-foundation/.

[11] "mogu_blog_v2," https://github.com/moxi624/mogu_blog_v2.

[12] "OpenFeign," https://spring.io/projects/spring-cloud-openfeign.

[13] "OWASP 2017," https://owasp.org/www-project-top-ten/2017/.

[14] "OWASP 2021," https://owasp.org/Top10/.

[15] "Prompt engineering." https://platform.openai.com/docs/guides/prompt-engineering.

[16] "RestTemplate," https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html.

[17] "Sitewhere," https://github.com/sitewhere/sitewhere.

[18] "Spring," https://spring.io/projects.

[19] "Spring-cloud-dataflow," https://github.com/spring-cloud/spring-cloud-dataflow.

[20] "Spring CVE," https://spring.io/security/cve-2024-22263.

[21] "Survey of Microservice Ecosystem 2021," https://www.jetbrains.com/lp/devecosystem-2021/microservices/.

[22] "Survey of Microservice Ecosystem 2022," https://www.jetbrains.com/lp/devecosystem-2022/microservices/.

[23] "The CWEs Supported by CodeQL," https://codeql.github.com/codeql-query-help/java-cwe/.

[24] "The CWEs Supported by CodeQL," https://github.com/github/codeql/tree/main/java/ql/src/Security/CWE.

[25] "The Definition of Monolithic Application," https://en.wikipedia.org/wiki/Monolithic_application.

[26] "The Official Repo of CNCF cloudevents," https://github.com/cloudevents/sdk-java.

[27] "The Routing Rules of Sring-cloud-gateway," https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway/gatewayfilter-factories.html.

[28] "The Routing Rules of Sring-cloud-gateway," https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway-server-mvc/gateway-handler-filter-functions.html.

[29] "The Sanitizer in CodeQL," https://github.com/github/codeql/blob/main/java/ql/lib/semmle/code/java/security/PathSanitizer.qll.

[30] "Yudao-cloud," https://github.com/YunaiV/yudao-cloud.

[31] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM sigplan notices*, vol. 49, no. 6, pp. 259–269, 2014.

[32] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and Flexible Discovery of PHP Application Vulnerabilities," in *2017 IEEE european symposium on security and privacy (EuroS&P)*. IEEE, 2017, pp. 334–349.

[33] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna, "Multi-Module Vulnerability Analysis of Web-based Applications," in *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)*, 2007, pp. 25–35.

[34] E. Bodden, "Inter-procedural data-flow analysis with IFDS/IDE and Soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, 2012, pp. 3–8.

[35] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language Models are Few-Shot Learners," *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS)*.

[36] R. Chandramouli, "Microservices-based Application Systems," *NIST Special Publication*, vol. 800, no. 204, pp. 800–204, 2019.

[37] M. Chen, T. Tu, H. Zhang, Q. Wen, and W. Wang, "Jasmine: A Static Analysis Framework for Spring Core Technologies," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[38] B. Davis and H. Chen, "DBTaint: Cross-Application Information Flow Tracking via Databases," in *USENIX Conference on Web Application Development (WebApps 10)*, 2010.

[39] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner," in *Proceedings of the 21st USENIX Security Symposium (USENIX)*, 2012.

[40] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox Data-driven Web Scanning," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[41] C. Fang, N. Miao, S. Srivastav, J. Liu, R. Zhang, R. Fang, R. Tsang, N. Nazari, H. Wang, H. Homayoun *et al.*, "Large Language Models for Code Analysis: Do LLMs Really Do Their Job?" in *Proceedings of the 33rd USENIX Security Symposium (USENIX)*, 2024, pp. 829–846.

[42] W. Gibbs, A. S. Raj, J. M. Vadayath, H. J. Tay, J. Miller, A. Ajayan, Z. L. Basque, A. Dutcher, F. Dong, X. Maso *et al.*, "Operation Mango: Scalable Discovery of Taint-Style Vulnerabilities in Binary Firmware Services," in *Proceedings of the 33rd USENIX Security Symposium (USENIX)*, 2024, pp. 7123–7139.

[43] N. Grech and Y. Smaragdakis, "P/Taint: Unified Points-to and Taint Analysis," vol. 1, no. OOPSLA. ACM New York, NY, USA, 2017, pp. 1–28.

[44] E. Güler, S. Schumilo, M. Schloegel, N. Bars, P. Görz, X. Xu, C. Kaygusuz, and T. Holz, "Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities," in *Proceedings of the 33rd USENIX Conference on Security Symposium (USENIX)*, 2024.

[45] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities," in *2006 IEEE Symposium on Security*

*and Privacy (SP)*, 2006.

[46] M.-A. Laverdière, B. J. Berger, and E. Merloz, "Taint analysis of manual service compositions using Cross-Application Call Graphs," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 585–589.

[47] O. Lhoták and L. Hendren, "Context-sensitive points-to analysis: is it worth it?" in *International Conference on Compiler Construction*. Springer, 2006, pp. 47–64.

[48] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen, "Automatic Policy Generation for Inter-Service Access Control of Microservices," in *Proceedings of the 30th USENIX Security Symposium (USENIX)*, 2021, pp. 3971–3988.

[49] Y. Liu, Y. Xue, D. Wu, Y. Sun, Y. Li, M. Shi, and Y. Liu, "PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation," *arXiv preprint arXiv:2405.02580*, 2024.

[50] J. Lu, H. Li, C. Liu, L. Li, and K. Cheng, "Detecting Missing-Permission-Check Vulnerabilities in Distributed Cloud Systems," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[51] C. Luo, P. Li, and W. Meng, "TChecker: Precise Static Inter-Procedural Analysis for Detecting Taint-Style Vulnerabilities in PHP Applications," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022, pp. 2175–2188.

[52] W. Ma, S. Yang, T. Tan, X. Ma, C. Xu, and Y. Li, "Context Sensitivity without Contexts: A Cut-Shortcut Approach to Fast and Precise Pointer Analysis," vol. 7, no. PLDI. ACM New York, NY, USA, 2023, pp. 539–564.

[53] F. Minna and F. Massacci, "SoK: Run-time security for cloud microservices. Are we there yet?" *Computers & Security*, vol. 127, p. 103119, 2023.

[54] F. Nielson and H. R. Nielson, "Infinitary Control Flow Analysis: a Collecting Semantics for Closure Analysis," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 1997, pp. 332–345.

[55] Y. Ouyang, K. Shao, K. Chen, R. Shen, C. Chen, M. Xu, Y. Zhang, and L. Zhang, "MirrorTaint: Practical Non-intrusive Dynamic Taint Tracking for JVM-based Microservice Systems," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2514–2526.

[56] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting Insecure Multi-binary Interactions in Embedded Firmware," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1544–1561.

[57] T. Reps, S. Horwitz, and M. Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995, pp. 49–61.

[58] E. Ruf, "Context-Insensitive Alias Analysis Reconsidered," *ACM SIGPLAN Notices*, vol. 30, no. 6, pp. 13–22, 1995.

[59] M. Sauwens, E. Heydari Beni, K. Jannes, B. Lagaisse, and W. Joosen, "ThunQ: A Distributed and Deep Authorization Middleware for Early and Lazy Policy Enforcement in Microservice Applications," in *Service-Oriented Computing: 19th International Conference, ICSOC 2021, Virtual Event, November 22–25, 2021, Proceedings 19*. Springer, 2021, pp. 204–220.

[60] T. Tan and Y. Li, "Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023, pp. 1093–1105.

[61] E. Trickel, F. Pagani, C. Zhu, L. Dresel, G. Vigna, C. Kruegel, R. Wang, T. Bao, Y. Shoshitaishvili, and A. Doupé, "Toss a fault to your witcher: Applying Grey-box Coverage-guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities," in *2023 IEEE symposium on security and privacy (SP)*, 2023.

[62] X. Zhou, T. Zhang, and D. Lo, "Large Language Model for Vulnerability Detection: Emerging Results and Future Directions," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2024, pp. 47–51.

## Appendix A. Assigned CVEs

Table 5 presents the CVEs assigned to us, along with their vulnerability types.

## Appendix B. Prompt Example

Figure 11 illustrates the prompts used by `MScan`. Following the 'Tactic: Provide examples' section in the best practices for prompt engineering provided by OpenAI [15], we have respectively set up SYSTEM, USER, and ASSISTANT roles. We assigned the system role to the task instructions, the user role to example queries in K shots and the actual query, and the assistant role to example answers in K shots. For simplicity, some examples and parts of lengthy code have been omitted.

## Appendix C. Inter-service Communication Mechanism

Table 6 provides details of the modeled APIs for inter-service communication components. These components include various commonly used communication components

TABLE 5: Assigned CVEs.

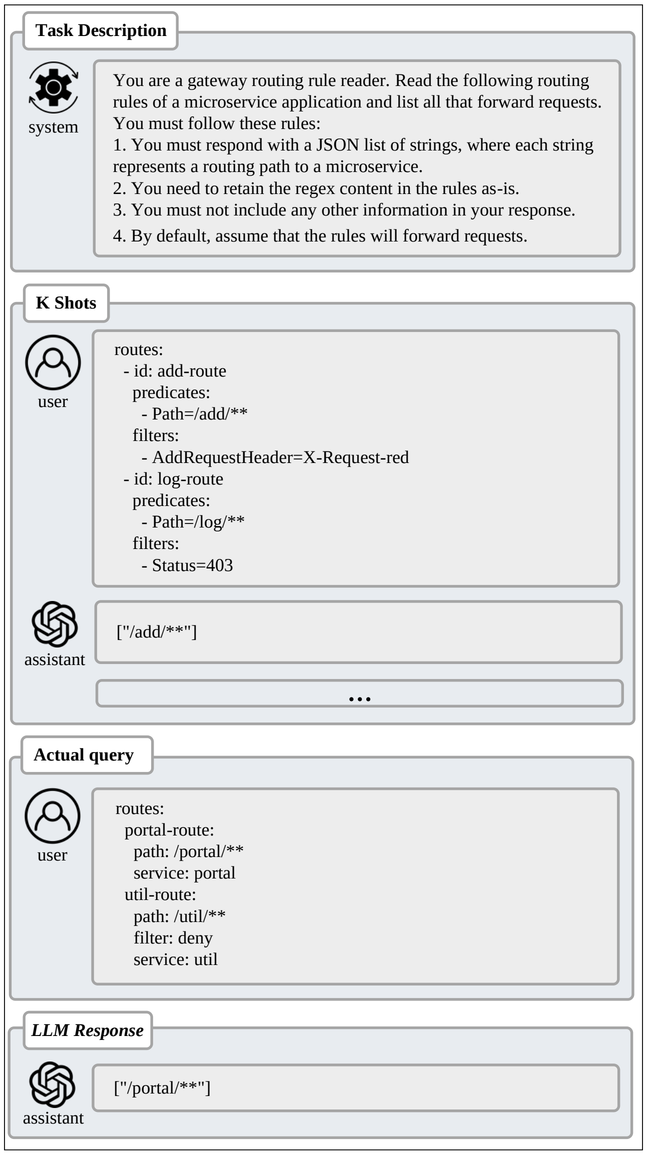
| Open-source Apps | CVEs[1] | Type[2] | Vuln |
|---|---|---|---|
| spring-cloud-dataflow | CVE-2024-22263 | AFW | Inter |
| yudao-cloud | CVE-2024-36686 | AFW | Inter |
| | CVE-2024-30738 | | Inter |
| | CVE-2024-30739 | | Intra |
| paascloud-master | CVE-2024-30743 | SQLi | Inter |
| | CVE-2024-30741 | | Intra |
| youlai-mall | CVE-2024-30742 | SQLi | Intra |
| lamp-cloud | CVE-2024-35596 | AFW | Inter |
| | CVE-2024-30762 | SQLi | Intra |
| | CVE-2024-30763 | | Intra |
| springblade | CVE-2024-30760 | SQLi | Intra |
| basemall | CVE-2024-35599 | AFR | Inter |
| microservices-platform | CVE-2024-32240 | AFR | Intra |
| mogu_blog_v2 | CVE-2024-35597 | SSRF | Inter |
| | CVE-2024-30955 | XXE | Inter |
| | CVE-2024-36690 | | Intra |
| open-mall | CVE-2024-35600 | AFW | Inter |
| | CVE-2024-35601 | | Inter |
| | CVE-2024-35602 | | Inter |
| | CVE-2024-35603 | | Inter |
| | CVE-2024-35604 | | Inter |
| | CVE-2024-35605 | | Inter |
| | CVE-2024-35606 | | Intra |
| | CVE-2024-35608 | | Intra |
| | CVE-2024-35611 | | Intra |
| | CVE-2024-30744 | | Intra |
| | CVE-2024-30745 | | Intra |
| | CVE-2024-30746 | SQLi | Intra |
| sitewhere | CVE-2024-37827 | SQLi | Inter |
| | CVE-2024-30754 | | Inter |
| | CVE-2024-30755 | | Intra |

[1] For ethical considerations, we anonymized the entire CVE identifiers.
[2] AFW: Arbitrary File Write; AFR: Arbitrary File Read; SQLi: SQL Injection; XXE: XML External Entity injection; SSRF: Server Side Request Forgery.

in microservice applications, including popular message queues and libraries in JDK.

## Appendix D. Case Study

Here, we additionally showcase some interesting inter-service taint-style vulnerabilities detected by MScan in highly popular real-world applications.



**Task Description**

system

You are a gateway routing rule reader. Read the following routing rules of a microservice application and list all that forward requests. You must follow these rules:
1. You must respond with a JSON list of strings, where each string represents a routing path to a microservice.
2. You need to retain the regex content in the rules as-is.
3. You must not include any other information in your response.
4. By default, assume that the rules will forward requests.

**K Shots**

user

```
routes:
  - id: add-route
    predicates:
      - Path=/add/**
    filters:
      - AddRequestHeader=X-Request-red
  - id: log-route
    predicates:
      - Path=/log/**
    filters:
      - Status=403
```

assistant

```
["/add/**"]
```

...

**Actual query**

user

```
routes:
  portal-route:
    path: /portal/**
    service: portal
  util-route:
    path: /util/**
    filter: deny
    service: util
```

***LLM Response***

assistant

```
["/portal/**"]
```

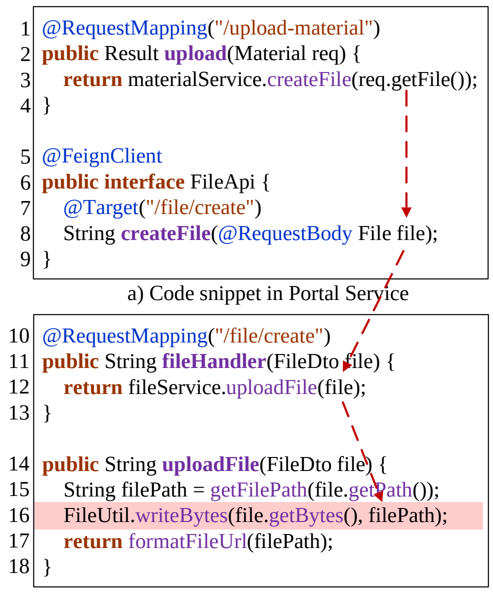
Figure 11: Example prompt and LLM response.

### D.1. AFW in Yudao-Cloud (Service Communication via OpenFeign [12])

The yudao-cloud [30] is an open-source and widely used development platform application, with over 15,000 stars on GitHub. As shown in Figure 12, MScan detected an arbitrary file write (AFW) vulnerability within the application that could lead to the server being taken over by an attacker. This vulnerability involves two microservices within the application, i.e., the `Portal` service and the `File` service. In the `Portal` service, the `upload` method serves as a user-accessible entry point, accessible via an HTTP request to the `/upload-material` path (lines 1-2). Then, the user input is passed to the `createFile` method through the method invocation (line 3). Specifically, the `createFile` method is decorated with the `@FeignClient` (line 5) and

TABLE 6: Communication modes supported by `MScan`.

| **Framework / Lib** | **Type** | **APIs** |
|---|---|---|
| OpenFeign | Sync | @FeignClient |
| RestTemplate | Sync | RestTemplate.get<br>RestTemplate.post<br>RestTemplate.exchange |
| gRPC | Sync | *ImplBase.*<br>*BlockingStub.* |
| JDK Native | Sync | URL.openConnection<br>HttpClient.send |
| Apache HttpClient | Sync | HttpClient.execute |
| Hutool-http | Sync | HttpUtil.get<br>HttpUtil.post |
| Dubbo | Sync | @DubboReference<br>@DubboService |
| Kafa | Async | KafkaProducer.send<br>KafkaConsumer.poll |
| RabbitMQ | Async | Channel.basicPublish<br>Channel.basicConsume |
| Redis | Async | Jedis.get<br>Jedis.set |
| MQTT | Async | MqttClient.publish<br>MqttClient.subscribe |



```
 1 @RequestMapping("/upload-material")
 2 public Result upload(Material req) {
 3   return materialService.createFile(req.getFile());
 4 }

 5 @FeignClient
 6 public interface FileApi {
 7   @Target("/file/create")
 8   String createFile(@RequestBody File file);
 9 }
```

a) Code snippet in Portal Service

```
10 @RequestMapping("/file/create")
11 public String fileHandler(FileDto file) {
12   return fileService.uploadFile(file);
13 }

14 public String uploadFile(FileDto file) {
15   String filePath = getFilePath(file.getPath());
16   FileUtil.writeBytes(file.getBytes(), filePath);
17   return formatFileUrl(filePath);
18 }
```

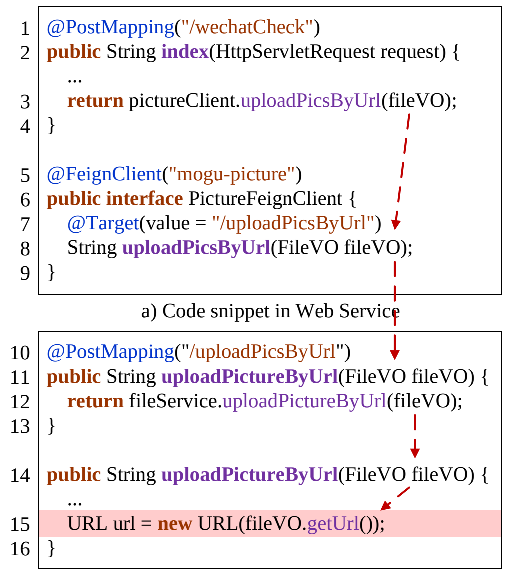
b) Code snippet in File Service

Figure 12: Arbitrary File Write vulnerability in Yudao-Cloud application (over 15k stars on Github).

`@Target` (line 7) annotations. The `@FeignClient` annotation indicates that the methods within this class are intended to invoke a method within another microservice, and the `@Target` annotation specifies the target address for this inter-service communication, i.e., the `fileHandler` method of the `File` service (line 11). Ultimately, the user input traverses through inter-service communication and reaches the sink method `writeBytes` on line 16 of the `File` service, resulting in an arbitrary file write vulnerability. Given the extensive potential damage posed by this vulnerability, we reported this critical issue to the developers and received a CVE (CVE-2024-36686).

### D.2. SSRF in mogu_blog_v2 (Service Communication via OpenFeign [12])

The mogu_blog_v2 [11] is a widely popular microservice-based blog application, earning over 1.6k stars on GitHub. Figure 13 illustrates an SSRF vulnerability identified by `MScan` in this application. This vulnerability spans the `Web` and `Picture` microservices. In the `Web` microservice, user input starts from `index` entry point (line 2) and propagates to the `uploadPicsByUrl` method (line 3), which subsequently interacts with the `Picture` microservice through OpenFeign (line 8). Within the `Picture` microservice, the input is passed to the `uploadPictureByUrl` method and eventually reaches the URL class (line 15), resulting in an SSRF vulnerability. As with other vulnerabilities, we immediately reported this issue to the developers and received a CVE identifier (CVE-2024-35597).



```
 1 @PostMapping("/wechatCheck")
 2 public String index(HttpServletRequest request) {
     ...
 3   return pictureClient.uploadPicsByUrl(fileVO);
 4 }

 5 @FeignClient("mogu-picture")
 6 public interface PictureFeignClient {
 7   @Target(value = "/uploadPicsByUrl")
 8   String uploadPicsByUrl(FileVO fileVO);
 9 }
```

a) Code snippet in Web Service

```
10 @PostMapping("/uploadPicsByUrl")
11 public String uploadPictureByUrl(FileVO fileVO) {
12   return fileService.uploadPictureByUrl(fileVO);
13 }

14 public String uploadPictureByUrl(FileVO fileVO) {
     ...
15   URL url = new URL(fileVO.getUrl());
16 }
```

b) Code snippet in Picture Service

Figure 13: SSRF vulnerability in mogu_blog_v2 application (over 1.6k stars on Github).

## Appendix E. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

### E.1. Summary

This paper introduces `MScan`, a static analysis framework designed to detect taint-style vulnerabilities in microservice-structured web applications. The methodology tackles key challenges in analyzing these applications by (1) leveraging an LLM-based approach to identify user-accessible entry points from gateway configurations, (2) constructing a service dependency graph (SDG) to model both inter- and intra-service data flows, and (3) employing a distance-guided, context-sensitive analysis to improve precision while reducing overhead. `MScan` is built on the Tai-e framework and is effectively evaluated on 30 microservice-based Java web applications.

### E.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Identifies an Impactful Vulnerability.
- Provides a Valuable Step Forward in an Established Field.

### E.3. Reasons for Acceptance

1) The paper presents `MScan`, the first tool specifically designed for vulnerability detection in Java-based microservices. While the submitted version does not include a link to a public repository, the authors have committed to releasing a prototype of `MScan` upon publication.
2) `MScan` identified 59 previously unknown vulnerabilities across 25 open-source and 5 commercial Java-based applications. Following responsible disclosure, 31 CVEs were assigned, demonstrating the tool's effectiveness in uncovering impactful security issues.
3) The paper advances taint-style vulnerability detection, addressing key challenges in analyzing Java-based distributed microservice applications. The approach tackles complex challenges in this domain by integrating LLM-based entry point identification, service dependency graph construction, and distance-guided context-sensitive analysis. An extensive ablation study provides strong empirical support for the proposed design choices.

### E.4. Noteworthy Concerns

1) The dataset is limited to 30 Java-based applications, which may not fully represent the broader microservice ecosystem. Some design choices may be tailored to this specific set of applications, potentially limiting the generalizability of the proposed solution to other platforms, programming languages, or microservice architectures.