Detecting Privilege Escalation in Polyglot Microservices via Agentic Program Analysis

Penghui Li
Columbia University
pl2689@columbia.edu

Hong Yau Chong
Columbia University
hc3661@columbia.edu

Yinzhi Cao
Johns Hopkins University
yinzhi.cao@jhu.edu

Junfeng Yang
Columbia University
junfeng@cs.columbia.edu

***Abstract*—Microservices are widely adopted in modern cloud systems due to their scalability and fault tolerance. However, microservice architectures introduce significant complexity in privilege and permission control, creating risks of privilege escalation where attackers can gain unauthorized access to resources or operations. Detecting such vulnerabilities is challenging due to complex cross-service interactions, polyglot codebases, and diverse privileged operations and permission checks. We present NEO, an agentic program analysis framework that combines large language models (LLMs) with classic program analysis to address these challenges. NEO leverages an LLM-based agent that dynamically generates analysis plans, adapts code search strategies, and validates semantics. We develop code search primitives that enable NEO to perform scalable and flexible code exploration across services and languages. We evaluated NEO on 25 open-source microservice applications spanning 7 programming languages and 6.2 million lines of code. NEO uncovered 24 zero-day privilege escalation vulnerabilities and achieved 81.0% precision and 85.0% recall on a ground-truth dataset. Compared to existing program analysis and agentic solutions, NEO demonstrated significant improvements in both detection accuracy and scalability. We further showcased NEO's extensibility by applying it to other application domains and vulnerability types, uncovering 18 additional zero-day vulnerabilities.**

1. Introduction

Microservice architectures have become the dominant paradigm for building large-scale distributed systems in the cloud [1]. By decomposing monolithic applications into loosely coupled, independently deployable services, microservices enable high scalability, fault tolerance, and rapid development [2], [3]. Major companies like Amazon [4] and Google [5] have evolved their internal systems to use microservices, often deployed in polyglot architectures. They also provide cloud platforms (*e.g.*, AWS, Google Cloud) that enable other organizations to adopt microservice architectures [6], [7].

However, the microservice architecture makes permission control inherently complex. On the one hand, modern microservice systems involve numerous principals, such as users, roles, groups, service accounts, *etc.* On the other hand, each service manages a wide range of service-specific resources and implements diverse privileged operations using different programming languages and frameworks. Authentication (authN) and authorization (authZ) checks for permission control are distributed across services. All these make it extremely challenging to ensure privileged operations are properly protected.

We illustrate this complexity using a real-world scenario in Figure 1. When the user Alice initiates a profile update to switch to the `developer` role, the request flows through three services. The Gateway service acts as the system's entry point and performs authentication by verifying Alice's credentials and routing her request to the appropriate backend. The UserProfile service handles user-specific updates and prepares the role-switching request. Finally, the UserMgmt service manages user accounts and executes the `setUserRole(role=dev)` operation to finalize the change. Beyond authentication, this operation actually requires authorization to verify whether Alice is eligible for this role switch. Such a check could be implemented in multiple places: at the Gateway before forwarding the request, at the UserProfile before invoking UserMgmt, or at the UserMgmt before executing the role change. In implementation, each service must decide which checks it should perform and which it can assume other services have already validated.

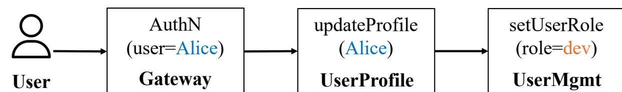


Figure 1: A user role update flows through three services.

When services fail to coordinate on authentication or authorization responsibilities, privilege escalation vulnerabilities arise. For instance, even if an authorization check validates that Alice can perform role changes, without verifying which specific roles she can assume, an attacker can manipulate the request to specify `role=admin` instead of `role=dev`. In fact, this is a realistic vulnerability we found in Mall4cloud [8], a popular cloud-based application with around 6K stars on GitHub. Such privilege escalations have been known with severe consequences, such as remote code execution [9], gaining root privileges [10], and compromising entire infrastructure [11].

Systematically detecting such privilege escalation vulnerabilities in polyglot microservice systems presents multiple

challenges. The first challenge lies in the inherent complexity of distributed, heterogeneous microservice systems (**C1**). As illustrated earlier, a single request often spans multiple services that are implemented using diverse programming languages and frameworks, and these services communicate through different protocols (*e.g.*, gRPC [12], REST [13], Kafka [14]). No individual service has a complete view of the end-to-end privilege enforcement logic, and thus requires analysis to holistically reason across service boundaries.

Second, it requires deep semantic understanding beyond structural code patterns (**C2**). Unlike memory corruptions (*e.g.*, buffer overflows) that can be characterized by code structure, finding privilege escalation requires recognizing which operations are privileged and which code implements authN/authZ checks. For example, determining that `setUserRole()` is a privileged operation demands semantic understanding of the function's purpose and security intent. Traditional program analysis approaches [15]–[19] rely on security experts to manually summarize such patterns and do not scale to diverse microservice implementations.

Third, even after identifying privileged operations and authN/authZ checks, it remains challenging to correlate which checks actually protect a given operation (**C3**). This requires analyzing the control and data dependencies between the operation and potential checks. These dependencies can traverse diverse structures ranging from simple conditional statements and function call sites to framework-specific mechanisms like decorators, middleware chains, or policy modules. For instance, determining what protects an endpoint may require tracing from the endpoint to a decorator, then to a router definition, and finally to the check implementation. Each application structures these relationships differently, where a fixed analysis pipeline does not suffice.

To the best of our knowledge, no prior work can solve these challenges. Existing microservice analysis tools [11], [20], [21] focus on other concerns and do not detect privilege escalation vulnerabilities. For example, MScan [20] targets taint-style vulnerabilities such as command injection in Java-based microservices but does not reason about privileges. Microscope [21] analyzes how code changes in one service impact others during development, but does not perform any security analysis. On the other hand, privilege escalation detection approaches developed for monolithic web [15], [16] and mobile [22] applications cannot handle polyglot microservices because they assume a single-language codebase with no cross-service interaction.

Our key insight is that large language models (LLMs) and classic program analysis have complementary strengths. LLMs excel at semantic reasoning about code intent, while classic program analysis provides scalable cross-service tracking. Combined through an agentic design, they enable dynamic correlation across complex code structures. For **C1**, we design code search primitives that abstract classic static analysis operations (*e.g.*, flow tracking, call-graph traversal) with language-agnostic interfaces using CodeQL [23], enabling scalable cross-service analysis across polyglot codebases. For **C2**, we leverage LLMs to interpret semantic cues from program identifiers and natural language artifacts (*e.g.*, function names, comments) to identify privileged operations and reason about security check adequacy. For **C3**, an LLM agent uses these primitives to trace control and data dependencies, and correlate operations and checks across different structural patterns.

We realize these ideas in a novel agentic program analysis framework, called NEO. NEO iteratively orchestrates code search primitives and LLM reasoning to identify privileged operations, trace cross-service flows, and locate and validate authN/authZ checks. A key novelty of NEO is that it treats code as structured program representations and leverages code search primitives to enable efficient, on-demand context retrieval from large polyglot codebases. Prior code agents [24]–[26] rely on LLMs to directly read and reason about entire files as unstructured text, which limits scalability and easily exhausts the context. While implementing our primitives atop CodeQL [23] inherits its inherent language-modeling limitations, this foundation ensures structural precision and cross-service scalability that are currently infeasible through purely text-based agent reasoning.

We extensively evaluated NEO on 25 open-source microservice applications, including an evaluation corpus of 21 applications and a ground-truth dataset of 4 applications. NEO successfully identified 24 zero-day privilege escalation vulnerabilities. On the ground-truth dataset, NEO achieves 81.0% precision and 85.0% recall. Compared to prior classic and agentic program analysis solutions, NEO demonstrates significant improvements in detection accuracy, scalability, and vulnerability coverage. For instance, compared to the EnIGMA agent that directly reads and analyzes entire source files, NEO found 24 more vulnerabilities. Our ablation study further shows that the code search primitives are essential: without them (*e.g.*, directly invoking CodeQL for context retrieval), the system missed most of the vulnerabilities. Beyond privilege escalation, we demonstrated NEO's extensibility by applying it to detect other vulnerability types across different application domains, uncovering 18 additional zero-day vulnerabilities. All vulnerabilities were responsibly disclosed to maintainers, with 8 fixed to date.

- We designed code search primitives that enable flexible code exploration and context retrieval.
- We developed NEO, a novel agentic program analysis framework that combines LLM semantic reasoning with scalable program analysis.
- NEO discovered a total of 42 new vulnerabilities and demonstrated superior performance over existing methods.

2. Background

2.1. Microservice Architecture

Microservice architectures decompose applications into multiple services that communicate over network protocols. A characteristic of microservices is their *polyglot nature*, where different services can be implemented in different programming languages and frameworks. For example, a

```java
@PreAuthorize("isAuthenticated()")
@PostMapping("/updateProfile")
public String updateProfile(@RequestBody ProfileRequest req) {
  String username = getUsername();
  String token = getToken();

  HttpHeaders headers = new HttpHeaders();
  headers.set("Authorization", "Bearer " + token);

  Map<String, Object> data = Map.of("username", username, "role",
  ↪  req.getRole());
  HttpEntity<Map<String, Object>> entity = new HttpEntity<>(data,
  ↪  headers);

  // request Python service to update role
  return restTemplate.postForObject(
  ↪  "http://localhost:5000/setUserRole", entity, String.class);
}
```

(a) UserProfile service written in Java.

```python
# add authorization in router dependency
router = APIRouter(dependencies=[Depends(authz)])

@router.post("/setUserRole")
async def setUserRole(request: Request):
  data = await request.json()
  update_role(data.get('username'), data.get('role'))
  return {"message": "Role updated successfully"}

def authz(request: Request):
  try:
    token = request.headers["Authorization"].split()[1]
    user = jwt.decode(token, "secret")["sub"]
    data = json.loads(await request.body())
    if not can_switch_roles(user):
      raise HTTPException(detail="Not eligible")
  except Exception:
    raise HTTPException(detail="Unauthorized")
```

(b) UserMgmt service written in Python.

Figure 2: Code implementation of the cross-service privilege escalation vulnerability in Figure 1. Both services perform authentication but fail to validate eligibility for the specific role being requested, allowing authenticated users to escalate to arbitrary roles.

data analytics service might use Python for its rich scientific libraries, while a high-performance API gateway uses Java or Go, and a real-time notification service leverages Node.js for asynchronous I/O. Services interact with each other through REST APIs [13], gRPC [12], GraphQL [27], and message brokers like Kafka [14] and RabbitMQ [28]. While this flexibility enables organizations to choose optimal technologies for each component, it introduces challenges for cross-service security analysis since traditional static analysis tools are typically single-language.

2.2. Permission Control and Privilege Escalation

Microservice applications often implement two distinct security mechanisms for permission control. *Authentication (authN)* verifies the identity of the requester by confirming "who you are" through credentials like passwords, tokens, or certificates. *Authorization (authZ)* determines what the authenticated user is permitted to do by verifying "what you can access" based on the user's identity and the requested resource. For example, in the scenario of role-switching, verifying Alice's credentials confirms authentication, but checking whether Alice is permitted to switch to the `developer` role confirms authorization. *Privilege escalation* occurs when a user gains access to resources or operations beyond what they are authorized to have, such as when security checks are missing, insufficient, or improperly implemented, allowing users to bypass intended access restrictions.

2.3. LLM-based Code Agents

LLM-based agents are autonomous systems that iteratively perceive their environment, reason about goals, and take actions through tool invocations to achieve objectives. Recent work has applied this paradigm to code analysis tasks [24]–[26], [29], [30], where agents interact with codebases through tools like file system operations and command execution. For example, SWE-agent [24] provides agents with bash commands to navigate repositories, such as listing directories with `ls`, searching keywords with `grep`, and reading file contents. Building upon SWE-agent, EnIGMA [26] extends the agent framework to cybersecurity tasks by introducing interactive tools that enable the LLM to use debuggers and other utilities essential for vulnerability analysis. These agents operate through a perception-action loop. The LLM observes the current state (*e.g.*, file contents, execution results), reasons about next steps, invokes tools to gather more information or make changes, and directly processes the returned outputs to guide subsequent decisions.

3. Motivation

3.1. Motivating Example

We present in Figure 2 simplified code for the vulnerability introduced in Figure 1. The two services involved in the vulnerability use Java and Python, respectively. The Java service in Figure 2(a) exposes an endpoint `/updateProfile` that accepts a role update request (line 2). The service enforces user authentication via the `@PreAuthorize()` annotation (line 1), verifying that the requester possesses valid credentials. Upon receiving the request, the service retrieves the username and token from the current session (lines 4-5), constructs an HTTP request with authorization headers (lines 7-8), and forwards the request to the Python service at `http://localhost:5000/setUserRole` (line 14).

The Python service in Figure 2(b) exposes an endpoint `/setUserRole` that receives the forwarded request (line 19). The service extracts `username` and `role` from the request body and directly updates the user's role (lines 22-23). Authentication is enforced at the router level, where the dependency `Depends(authz)` (line 17) validates the JWT token's authenticity (lines 27-28). However, the `authz()` function is insufficient as it checks whether the user has the general permission to switch roles via `can_switch_roles()`, but fails to validate whether the user is eligible for the specific role being requested, *e.g.*, `dev` or `admin`. This creates a severe

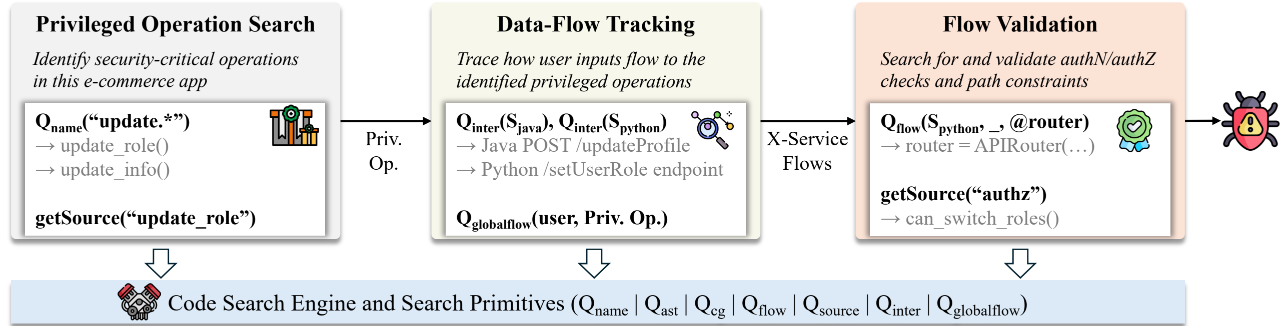


Figure 3: Workflow of NEO. The LLM agent uses code search primitives to iteratively identify privileged operations, trace cross-service flows, and validate security checks.

privilege escalation vulnerability, allowing any authenticated user to assume higher-privileged roles.

3.2. Challenges

We revisit the three challenges introduced in §1 using the code of the example.

C1: Distributed and Heterogeneous Architecture. The vulnerability in Figure 2 spans Java and Python services communicating via REST. Detecting it requires identifying the inter-service call at line 14, tracing how the user-controlled `role` parameter propagates across the service boundary to `http://localhost:5000/setUserRole`, and verifying authorization in either service. Traditional static analysis tools cannot perform such cross-language data flow analysis [15], [17], [20].

C2: Semantic Understanding. Recognizing that `setUserRole()` is a privileged operation requires understanding the function's security-critical purpose. Moreover, determining whether the service-specific `authz()` provides adequate protection requires reasoning about what `can_switch_roles()` actually verifies—whether it checks general role-switching permission or validates eligibility for the specific role requested. This semantic gap cannot be captured by manually enumerated patterns.

C3: Correlating Checks and Operations. Determining which checks protect `setUserRole()` in Figure 2(b) requires correlating the operation with security checks through control and data dependencies. From the endpoint decorator (line 19), the analysis must trace to the router definition (line 17), discover the `Depends(authz)` dependency, and retrieve its implementation (lines 25-33) to establish the correlation. Different applications structure these relationships differently, requiring the analysis to adapt rather than follow a predetermined pipeline.

4. Design

In this section, we present the design of NEO, our agentic program analysis framework for detecting privilege escalation vulnerabilities in polyglot microservices. NEO addresses the challenges through the orchestrated interplay of LLM reasoning and static program analysis.

The overall architecture of NEO is illustrated in Figure 3. To detect privilege escalation vulnerabilities, we decompose the analysis into three interconnected components. First, NEO iteratively identifies privileged operations by analyzing function names, documentation, comments, and source code to determine which operations require authorization checks. Second, NEO conducts holistic cross-service data-flow analysis to trace how data propagates from external user inputs to the identified privileged operations. Since attackers can manipulate user-controlled inputs, this analysis identifies potential attack vectors where malicious requests could reach privileged operations. Third, NEO traverses the identified flows to locate authN/authZ checks along these paths. When needed, NEO dynamically retrieves additional context to validate whether the checks verify appropriate properties (*e.g.*, role eligibility vs. mere authentication). NEO also checks the feasibility of the paths using a lightweight path constraint collection and satisfiability checking.

From a high level, NEO takes as input the source code of a microservice application and a prompt that describes the detection task. The output is a set of privilege escalation vulnerability reports. To enable this workflow, we design a code search engine (§4.1) that provides the building blocks for NEO to navigate codebases and perform program analysis. Next, we describe privileged operation identification in §4.2, explain cross-service flow tracking in §4.3, and present security check validation in §4.4.

4.1. Code Search Engine

We frame static program analysis as a code search problem over the target application's codebase, where vulnerability detection logic is expressed as code queries [23], [31]. Formally, given an application A, a code search query $Q(A)$ returns matching code components when the query is executed. Complex program analysis can be composed by chaining multiple searches, where subsequent queries operate on the results of preceding queries. An LLM agent can dynamically plan and adapt its search strategy by composing these code search primitives based on intermediate results.

TABLE 1: Fundamental query categories and their frequencies across 290 CodeQL queries.

Symbol	Category	Description	Example	Frequency (%)
Q_{name}	Name-based lookup	Match code elements using identifier names or keywords	Find `update_role()`	286 (98.6%)
Q_{ast}	AST-based lookup	Identify code patterns by analyzing AST structures	Find all method calls	214 (73.8%)
Q_{flow}	Flow tracking	Track how values propagate between program elements	Trace `request` to sink	205 (70.7%)
Q_{cg}	Call-graph traversal	Explore relationships between functions through the call graph	Find callers of `authz()`	144 (49.7%)

Prior query-based solutions such as Joern [31] and CodeQL [23] have been highly successful and widely adopted in practice. They provide expressive domain-specific query languages that offer human developers fine-grained control and flexibility to specify complex security queries. However, this expressiveness comes with complexity that hinders its use by LLM agents. The large syntax space and extensive API surface (*e.g.*, several thousand APIs in Joern and CodeQL) make these query languages challenging for LLMs to learn and generate correctly, especially given their limited resources in LLM pretraining data. Moreover, these tools require different query syntax for the same semantic concept across programming languages. For instance, querying method calls in CodeQL requires `MethodAccess` for Java but `Call` for Python, each with different properties and predicates to access callers, arguments, and return values. This complexity leads to frequent errors when LLMs attempt to generate queries, such as using incorrect syntax or mixing constructs across languages. In an agentic design that requires multiple rounds of iterative querying, such errors become particularly costly, as each mistake requires a great number of additional LLM calls to correct and retry (see §6.3 for details).

To address these challenges, we design the code search engine with a small set of *code search primitives* to enable scalable and flexible code search (§4.1.1). We also present a set of property functions for obtaining detailed information from code components (§4.1.2).

4.1.1. Code Search Primitives. To identify the core query operations needed for static security analysis, we examined how existing CodeQL queries detect vulnerabilities. Specifically, we analyzed 290 security detection queries in the "Security" folder of CodeQL's query library (version 2.22.4) [32] for C++ and Java. Through manual inspection, we identified four fundamental query operations. These primitives represent different levels of reasoning over program structures, from simple text matching to interprocedural analysis. Table 1 summarizes their usage frequencies across the analyzed queries. Name-based lookup (Q_{name}) matches code elements by identifier names or keywords to locate functions, variables, or types (98.6%). AST-based lookup (Q_{ast}) captures structural patterns in the program's abstract syntax tree (73.8%). Flow tracking (Q_{flow}) traces how values propagate between sources and sinks (70.7%). Call-graph traversal (Q_{cg}) analyzes interprocedural relationships, such as identifying callers or callees of specific functions (49.7%).

These observations reveal that, despite differences in language and vulnerability type, most analyses rely on a small set of recurring query patterns. We therefore design NEO's code search engine around these four unified *code search primitives*. Each primitive abstracts away language-specific syntax and is exposed through a *unified API* that LLM agents can invoke consistently across programming languages. Unlike prior systems such as CodeQL, which require thousands of language-specific predicates and class hierarchies, our design provides a concise, semantically aligned interface. This simplification allows LLMs to compose complex analyses through simple, chainable queries without memorizing tool-specific syntax. By offering a small yet expressive set of primitives, NEO reduces query-generation errors while preserving the analytical depth required for comprehensive vulnerability detection.

Name-based Lookup $Q_{name}(service, name)$. This query locates program elements (functions, classes, variables, *etc.*) in a service by their identifier names. It is the most direct means of identifying security-critical code elements and retrieving on-demand context. The operation supports both exact matching and regular expression patterns for fuzzy matching. For example, $Q_{name}(service, \text{"}update_role\text{"})$ locates the exact operation `update_role()` that modifies user roles, while $Q_{name}(service, \text{"}update.*\text{"})$ finds all functions whose names start with "update". This operation is also useful for identifying framework-specific handlers. For instance, $Q_{name}(service, \text{"}.*Router\text{"})$ locates FastAPI router definitions, and $Q_{name}(service, \text{"}auth.*\text{"})$ finds authorization-related functions like `authz()`.

AST-based Lookup $Q_{ast}(service, operation)$. This query identifies code locations that perform specific syntactic operations, such as field accesses, binary operations, or variable assignments. We define a unified vocabulary of operation types that cover common code patterns across languages. For example, $Q_{ast}(service, \text{"}call\text{"})$ locates all function/method invocations, regardless of whether they are represented as `MethodAccess` in Java, `Call` in Python, or `CallExpr` in JavaScript. Similarly, $Q_{ast}(service, \text{"}field_access\text{"})$ finds all field or attribute accesses (*e.g.*, `obj.field`). By providing a language-agnostic interface for common syntactic patterns, this operation enables the LLM agent to search for specific code constructs without learning the different AST node types in each language's analysis tooling.

Flow Tracking $Q_{flow}(service, from, to)$. This query traces how data propagates from one program element to another within a service. For example, $Q_{flow}(service, \text{"}request\text{"}, \text{"}update_role\text{"})$ traces how the user-provided `role` parameter flows in Figure 2(b). The operation returns the complete path at variable-level granularity, showing data propagation as `request` $\rightarrow$ `data.get("role")` $\rightarrow$ `update_role(username, role)`. This enables the agent to identify that user-controlled data reaches a privileged operation.

Call Graph Traversal $Q_{cg}(service, function, direction)$**.** This query provides bidirectional call graph navigation in a service, finding all callers of a given function or all callees it invokes. For example, $Q_{cg}(service, \text{"}update_role\text{"}, \text{"}callers\text{"})$ identifies all code paths leading to the privileged operation, while $Q_{cg}(service, \text{"}setUserRole\text{"}, \text{"}callees\text{"})$ reveals what operations a handler performs. The agent can iteratively traverse the call graph to discover execution paths and verify security checks are present. Note that precisely identifying call targets, especially for dynamic calls, remains an open challenge in static analysis. Our implementation relies on CodeQL's call resolution, which provides sound approximations but may be conservative in complex scenarios involving reflection or dynamic dispatch.

4.1.2. Property Functions. For elements returned by query operations, we define a set of property functions that provide rich information to enable the LLM agent to reason about the elements. Three property functions are particularly important.
getLocation(). This function returns the element's location in the codebase, including file path, line number, and column position. This allows the agent to precisely reference code locations when reporting vulnerabilities or retrieving surrounding context. For example, after identifying a privileged operation, the agent can use `getLocation()` to determine which service and file it resides in.
getSource(). This function returns the source code of the element, ranging from a single statement to an entire function body. This enables the agent to examine the actual implementation without additional queries. For instance, after locating `update_role()` via Q_{name}, the agent can call `getSource()` to analyze its implementation and determine whether it performs authorization checks.
getType(). This function returns the type of the element. Specifically, it performs type inference for variables to determine their types. This is helpful when filtering query results based on value types or for identifying sanitizations where type conversions occur.

4.2. Finding Privileged Operations

In this work, we define privileged operations as call sites that access sensitive resources (*e.g.*, user-specific records, configuration settings), perform security-critical actions (*e.g.*, database writes, system commands), or modify protected state (*e.g.*, permissions, authentication tokens, user roles). To identify privileged operations, we provide the privileged operation definition in the prompt, along with instructions for using the code search engine, and instruct the LLM to generate queries to search the codebase. The agent iteratively queries the codebase, validates results via an LLM, and outputs the locations of privileged operations. Optionally, NEO analyzes application documentation (*e.g.*, README files) to understand the application's purpose and generate more targeted queries. For example, after understanding the motivating example is a user management system, NEO generates queries like $Q_{name}(service, \text{"}update.*\text{"})$ to locate state modification operations and $Q_{name}(service, \text{"}.*role.*\text{"})$ to find role-related functions. These regular expression queries allow fuzzy matching, discovering operations like `update_role()`, `setUserRole()`, and `assign_role()`. NEO can also use Q_{ast} queries to find specific operation types, such as $Q_{ast}(service, \text{"}method\text{"})$ to locate all methods (functions) defined in the service.

Because queries may match many sites beyond actual privileged operations, NEO must validate each candidate. For each candidate, NEO retrieves the source code using `getSource()` and then consults the LLM to assess whether this is a privileged operation, classifying it accordingly based on the provided definitions. For instance, when validating `update_role()`, NEO observes that it modifies user roles in the database, a security-critical action that changes user privileges. The process continues with additional query rounds if the iteration limit has not been reached, allowing NEO to refine its search based on discovered patterns.

4.3. Identifying Cross-service Flows

After identifying privileged operations, NEO must determine which operations are reachable from external user inputs (*i.e.*, there is a data flow path), as only reachable operations can be exploited by attackers for privilege escalation. The code search primitives described earlier operate within individual services. Privilege escalation vulnerabilities in microservices often span multiple services, requiring analysis to trace data flow across service boundaries. While the agent could in principle chain the primitives to perform cross-service analysis, this would require complex multi-step reasoning to identify service entry points, match inter-service calls to their target endpoints, and stitch together flows across service boundaries. Such a process is often prone to errors and inefficiency. We thus define three additional operations to facilitate cross-service privilege escalation detection.
Source Identification $Q_{source}(service)$**.** This query identifies data sources in a service, *i.e.*, locations where untrusted data from external sources enters the service (*e.g.*, HTTP request parameters, message queue consumers, API endpoints). We also define Q_{user}—a special case of Q_{source}, which identifies external user inputs entering the entire system, typically at the proxy or gateway service. We reused the method in MScan [20] to identify user sources by prompting the LLM to analyze the gateway configuration file.
Inter-service Communication $Q_{inter}(service)$**.** This query identifies inter-service communication points, including outgoing HTTP calls, RPC invocations, Kafka messages, WebSocket connections, GraphQL queries, *etc.* These represent points where data flows from one service to another. Each inter-service call is identified by a unique channel identifier that associates the caller and callee. This design is inspired by prior work [20]. Consider the example in Figure 2. The outgoing HTTP call in the Java service uses the URL `http://localhost:5000/setUserRole` as its channel identifier, which NEO matches to the corresponding endpoint `/setUserRole` in the Python service. This enables NEO to connect flows across the language/service boundary.

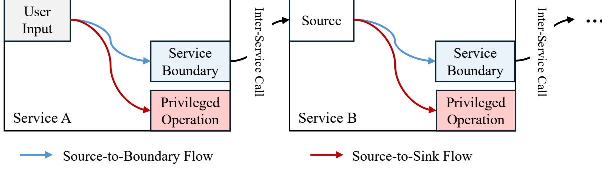


Figure 4: Illustration of inter-service analysis.

Global Flow $Q_{globalflow}(source, sink)$. This query traces data flow across multiple services from a source element to a sink element, connecting intra-service flows through inter-service communication points. Unlike Q_{flow} that operates within a single service, $Q_{globalflow}$ constructs a global reachability graph.

As illustrated in Figure 4, a data source entering a service can flow through two types of paths: (1) a source-to-boundary flow that reaches the service boundary to another service via an inter-service call, and (2) a source-to-sink flow that directly reaches a privileged operation. Inter-service calls act as bridges, connecting flows across service boundaries. $Q_{globalflow}$ operates in two phases. It first performs intra-service flow analysis using Q_{flow} to track these paths, and then chains them into a global inter-service view by matching inter-service calls identified by Q_{inter} to their target endpoints identified by Q_{source}.

Algorithm 1 presents the algorithm implementing $Q_{globalflow}$, which takes as input the set of services $\mathcal{S}$ and identified privileged operations $\mathcal{P}$, and outputs a global reachability graph $\mathcal{G}$. In the first phase, for each service s, the algorithm identifies user input entry points using $Q_{source}(s)$ (line 4), privileged operations $\mathcal{P}$ within the service (line 5), and outgoing inter-service calls using $Q_{inter}(s)$ (line 6). For each user input source, the algorithm uses $Q_{flow}(s, src, dst)$ to check if data can flow to privileged operations or inter-service calls (line 10). If a flow exists, an edge is added from source to sink in the global graph $\mathcal{G}$. In the second phase, the algorithm connects flows across service boundaries by linking each inter-service call to its target endpoint in the receiving service (lines 16-17). This leverages the approach from MScan [20], which assigns each inter-service communication channel a unique identifier to correlate outbound calls in one service with inbound data in the other service. Finally, $Q_{globalflow}$ returns all paths from external user inputs (Q_{user}) to privileged operations $\mathcal{P}$. These flows represent potential authorization vulnerabilities where user-controlled data can reach security-critical operations across service boundaries.

With these operations, NEO can identify global cross-service flows that can reach the privileged operations.

Algorithm 1: Inter-service flow analysis.

```
   Input  : Services 𝒮, Privileged operations 𝒫
   Output : Global reachability graph 𝒢
 1 𝒢 ← ∅
 2 // Phase 1: Intra-service flow analysis
 3 foreach s ∈ 𝒮 do
 4     Srcs ← Q_source (s)
 5     PrivOps ← {p ∈ 𝒫 | p ∈ s}
 6     InterCalls ← Q_inter (s)
 7     // Check reachability and add edges
 8     foreach src ∈ Srcs do
 9         foreach dst ∈ PrivOps ∪ InterCalls do
10             if Q_flow(s, src, dst) ≠ ∅ then 𝒢.addEdge(src,
                 dst) // Add path to G ;
11         end
12     end
13 end
14 // Phase 2: Inter-service connection
15 foreach inter-service call c : s1 → s2 in 𝒢 do
16     endpoint ← c.targetEndpoint in s2
17     𝒢.addEdge(c, endpoint) // Cross-service edge
18 end
19 return 𝒢
```

4.4. Validating Flows

NEO validates whether appropriate authN/authZ checks exist along these flows and whether the paths are feasible. This validation leverages the LLM's reasoning capabilities combined with *on-demand context retrieval* to handle diverse authN/authZ patterns across different frameworks and implementation styles. The agent traverses flow paths node by node (*e.g.*, function calls, conditionals), deciding at each point whether current information suffices or whether additional context is needed. At any point, the agent can flexibly issue code search primitives or property functions to retrieve specific code snippets across the entire codebase. Unlike prior code agents [24]–[26] that directly load entire source files into the LLM context, NEO reads only the structured results returned by search operations—such as function signatures, call relationships, or targeted code snippets. This abstraction achieves *scalability* and *efficiency* by avoiding context window exhaustion and enabling cross-language navigation through language-agnostic search results.

AuthN/AuthZ Check Localization. NEO traverses each flow to locate authN/authZ checks by analyzing decorators, middleware, or inline validation logic. For example, in Figure 2(b), when encountering `@router.post("/setUserRole")` (line 19) without explicit authorization, NEO queries $Q_{flow}(service, _, \text{"@router"})$ to locate the router definition (line 17) and discovers the `Depends(authz)` dependency. To understand what this dependency does, NEO retrieves `authz`'s implementation (lines 25-33) via Q_{name} and `getSource()`, confirming it validates JWT tokens. This iterative retrieval enables NEO to discover authentication mechanisms regardless of their locations in the code structure.

Sufficiency Assessment. Beyond locating checks, NEO must determine whether they adequately protect the privileged operation—a fundamentally semantic reasoning task. The agent performs this analysis by consulting the LLM with both the check's implementation and the privileged operation's context. The agent first asks the LLM to classify the check as authN or authZ, and if authZ, to identify its specific type (*e.g.*, role-based, permission-based, resource ownership). The agent then examines what protection the privileged operation

requires and identifies any semantic gap between what is checked and what is required. For instance, the insufficient authZ in Figure 2(b) could be correctly labeled by leveraging LLM's semantic understanding of security properties.

Path Constraint Validation. Static analysis typically cannot determine whether a flow is actually feasible or reachable, resulting in false positives. NEO mitigates this with a conservative path constraint validation stage to filter out infeasible paths. Specifically, NEO leverages the LLM to extract and collect path constraints from the identified flows, and prunes flows with unsatisfiable constraints. For each flow, the LLM traverses intermediate nodes in the flow and identifies conditional guards that constrain the path. The LLM then translates these constraints into logical predicates in SMT-LIB format, which are then sent to the Z3 solver [33] for satisfiability checking. If the Z3 solver reports unsatisfiable constraints, NEO marks it as a false positive.

Since the path constraints are collected by the LLM, they may introduce inaccuracies. For example, the generated constraints might be syntactically invalid or simply incomplete. We thus instruct the LLM to be *conservative*. If the LLM determines that the code logic is too complex to be reliably modeled as a constraint (*e.g.*, multiple layers of data dependencies), it skips this step and retains the uncertain cases as potential vulnerabilities.

5. Implementation

We implemented a prototype of NEO using 5.8K lines of CodeQL queries and 4.3K lines of Python code. NEO supports the 7 most popular programming languages for cloud applications: Go, Java, JavaScript (JS), Python, C#, C, and C++. We make NEO and the prompts available at https://github.com/columbia/neo. We next present several important implementation details.

Agent Orchestration. NEO implements the agentic loop in Python, coordinating LLM reasoning with program analysis primitives. The agent has access to a set of utility tools, including privileged operation search, data-flow analysis, call graph traversal, context retrieval, and validation tools. Each tool is realized either through carefully designed prompts that consult the LLM or through CodeQL queries. At each iteration, the LLM receives the current analysis state and generates the next action by executing a search primitive, requesting context, or performing validation. The agent stops when no new privileged operations are found, all flows are validated, or an iteration limit is reached.

NEO's task prompt describes the three-step workflow (Figure 3) and exposes the search primitives as well as the fallback option of bash commands to the LLM. We incorporate the application's documentation to allow the agent to tailor its analysis to each application. Two true vulnerabilities are also provided as demonstrations.

Code Database Construction. NEO builds upon CodeQL's infrastructure to represent the target application's source code in a structured database. The database captures multiple program representations, including abstract syntax trees for syntactic structure, control-flow graphs for execution paths, data-flow graphs for tracking data dependencies, and call graphs for inter-procedural relationships. For compiled languages like Java, the extraction analyzes compilation artifacts and intermediate representations. For interpreted languages like Python, the extraction directly parses source code to construct the database. Database construction is performed once per application.

Communication Channel Identification. We implemented Q_{inter} by developing comprehensive, language-specific CodeQL queries to identify communication patterns on both the caller and callee sides. To extract the channel identifier for each cross-service communication point, our implementation performs backward data-flow analysis from the communication call site to identify the string literal or constant defining the channel (*e.g.*, URL, topic name). Our implementation currently supports major protocols including REST, gRPC stubs, Kafka, RabbitMQ, WebSocket, and Dubbo.

CodeQL-based Search Primitives. We implemented the code search primitives as CodeQL templates with a few placeholders and developed Python scripts to dynamically fill them. When NEO issues commands like $Q_{flow}(source, sink)$, NEO dynamically populates these templates with the specific method names or types identified by the LLM. Each primitive is implemented as a separate CodeQL module that abstracts language-specific CodeQL predicates into a unified interface. For example, the Q_{ast} primitive for method calls translates to `MethodAccess` in Java, `Call` in Python, and `CallExpr` in JavaScript, but presents a single API to the agent. This design ensures that the LLM agent can query codebases without knowledge of language-specific analysis constructs.

SMT-LIB Constraint Generation. To generate the constraints for path validation, we provide the identified flows to the LLM and instruct it to convert the conditional statements (*e.g.*, `if`, `switch`) directly into SMT-LIB format. Specifically, similar to recent work [30], [34], [35], the LLM is prompted to declare relevant variables, define constants for status codes or fixed values, and assert the logical predicates encountered along the path. This process relies on the LLM's semantic understanding to interpret diverse coding patterns (*e.g.*, string comparisons or status flag checks) that are often difficult for traditional symbolic execution to model across different languages.

6. Evaluation

We extensively evaluate NEO to answer the following research questions:

- **Effectiveness.** Can NEO detect previously unknown vulnerabilities in real-world microservice applications?
- **Ablation Study.** What is the contribution of each component of NEO to its overall effectiveness?
- **Comparison with Prior Work.** How does NEO perform compared to state-of-the-art analysis tools?
- **Efficiency and Cost.** What are the runtime and LLM costs of running NEO?
- **Extensibility.** How applicable is NEO to other vulnerability analysis tasks?

TABLE 2: Testing dataset of 21 microservice applications. `# Stars` denotes the number of GitHub stars.

App	# Stars	# LoC	Languages
light-reading-cloud [36]	1,465	6,394	Java
PiggyMetrics [37]	13,757	19,942	JS, Java
Pitstop [38]	1,145	96,576	JS, C#
SiteWhere [39]	1,034	40,764	Java
Supermarket [40]	2,087	167,169	Java
Food Delivery [41]	941	178,063	JS, C#
Online Boutique [42]	19,254	31,179	JS, Java, Go, Python, C#
Booking [43]	1,256	34,974	JS, C#
Hotel Map [44]	1,085	2,696	Go
JBone [45]	1,008	12,360	Java
Train Ticket [46]	836	344,119	JS, Java, Go, Python
Mall4cloud [8]	5,933	116,043	JS, Java
AspnetRun E-Shop [47]	3,121	62,105	C#
Cinema [48]	1,773	8,152	JS
TODO App [49]	1,409	16,240	JS, Java, Go, Python
ABP eShopOnAbp [50]	736	220,134	JS, C#
Spring Boot Basics [51]	728	5,046	JS, Java
Magda [52]	564	625,045	JS, Java
Genie [53]	1,756	127,889	JS, Java, Python
DeathStarBench [54]	871	3,621,236	JS, C, C++, Go, Python
Swarm [55]	12,673	133,793	Java

6.1. Experimental Setup

Dataset. To comprehensively evaluate NEO's performance, we use two datasets: (1) an evaluation corpus of 21 open-source microservice applications, and (2) a ground-truth dataset containing 20 verified privilege escalation vulnerabilities across 4 applications.

Evaluation corpus. We constructed an evaluation corpus of 21 microservice applications by searching open-source repositories from GitHub with the keyword "microservice". Since several thousand repositories were returned in the search, we applied multiple filtering strategies to ensure quality and representativeness. First, each application must have at least 500 stars, indicating significant community adoption and popularity. Second, applications must be actively maintained, defined as having commits or releases within the past three years, to reflect current development practices. We also prioritized applications used in prior microservice research [11], [20] to enable direct comparison with existing work. Table 2 presents the dataset details, including lines of code (LoC) for each application. These applications span a wide range of complexity, from systems with a few services to large-scale benchmarks with dozens of services. They exhibit significant diversity in both application domains (*e.g.*, e-commerce, ticketing) and programming languages (*e.g.*, Java, Go, Python, C#), with most being polyglot systems.

Ground-truth dataset. We constructed a ground-truth dataset of applications with verified privilege escalation vulnerabilities to assess false negatives. Specifically, we searched MITRE CVE [56] and NVD [57] using the application names and identified previously reported privilege escalation vulnerabilities in these applications. The dataset includes 18 previously reported vulnerabilities across 4 applications, as well as 2 previously unreported vulnerabilities uncovered by NEO, yielding 20 verified vulnerabilities in total. Details are shown in Table 3.

TABLE 3: Ground-truth dataset with 20 verified vulnerabilities across 4 applications, including 18 previously reported vulnerabilities and 2 previously unreported vulnerabilities uncovered by NEO in Newbee Mall during evaluation.

App	# Stars	# LoC	Languages	# Vuls.
Newbee Mall [58]	11,437	172,057	Java	11 + 2
ZLT Platform [59]	4,721	36,950	Java, JS	3
Armeria [60]	5,033	633,902	Java, JS	2
Spring-cloud-dataflow [61]	1,138	235,572	Java, JS, Python	2

Evaluation Procedures. To run NEO, we first compile each application into a code database on which we can apply code search primitives. We then launch NEO and instruct it to autonomously detect privilege escalation vulnerabilities with a time budget of 10 hours per application. By default, we use Claude Sonnet 3.7 for agent reasoning with a temperature of 0.2. As for the in-context examples, we selected two vulnerabilities from applications outside the evaluation dataset to ensure the model generalizes its security reasoning. These examples were used across all experiments to provide consistent guidance. The experiments were conducted on a CPU server with two AMD EPYC 7502 32-Core processors and 251GB of RAM running Ubuntu 24.04.3 LTS.

6.2. Effectiveness

We first evaluated the effectiveness of NEO in detecting privilege escalation vulnerabilities and summarized the detection results in Table 4. In total, NEO identified 39 true-positive vulnerabilities out of 51 reports across the two datasets. 24 out of the 39 are new privilege escalation vulnerabilities. Specifically, in the evaluation corpus, NEO uncovered 22 previously unknown vulnerabilities. In the ground-truth dataset, NEO found 17 vulnerabilities in total, including 2 previously unreported vulnerabilities; this results in a recall of 85.0% with a precision of 81.0%. For all the reported cases from NEO in both datasets, we verified the vulnerabilities by manually analyzing the vulnerable flows from application entry to the privileged operations. We constructed proof-of-concept exploits to demonstrate the exploitability of these vulnerabilities. Overall, there were a total of 12 false positives generated, achieving an overall precision ($\frac{TP}{TP+FP}$) of 76.5%.

Characterization of True Positives. During the verification process, we found that these vulnerabilities showcase diverse security impacts, mostly determined by the variety of privileged operations. Standard ones include database operations and file operations, which can be exploited to perform SQL injection, arbitrary file reads and writes, or path traversal attacks. We also identified 18 application-specific privileged operations. These application-specific operations include privilege elevation regarding user roles or permissions, administrative command execution (*e.g.*, system configuration changes), and critical business logic operations (*e.g.*, payment processing or order fulfillment). This diver-

TABLE 4: Privilege escalation detection results. In the evaluation corpus, all true positives are previously unknown vulnerabilities. In the ground-truth dataset, NEO reports 17 true positives in total, including 2 previously unreported vulnerabilities.

Dataset	TP	FP	FN	Prec.	Recall
Eval. corpus	22	8	N/A	73.3%	N/A
Ground-truth dataset	17	4	3	81.0%	85.0%
Total	39	12	N/A	76.5%	N/A

sity demonstrates that privilege escalation vulnerabilities in microservices extend beyond traditional attack vectors and require a comprehensive analysis of application-specific security contexts. They are hard to (automatically) identify without NEO's LLM-based reasoning. We further present a case study in §6.7.

False Positives. We systematically analyzed the 12 false positives to understand their root causes and identified two main categories. First, 8 cases resulted from misidentifying the execution context, leading to over-approximation of security impacts. Specifically, NEO incorrectly classified client-side operations as server-side vulnerabilities. User-facing microservice applications often contain components that execute in the user's browser, such as browsing local directories for file uploads or displaying user-controlled content. While NEO correctly identified that user input influenced these operations, they actually execute client-side and are legitimately controlled by users themselves. The security impact is confined to the user's local environment with no server-side privilege escalation risk. This confusion arose because NEO failed to distinguish between client-side and server-side execution contexts. Second, the remaining 4 false positives resulted from NEO failing to detect authN/authZ checks. While NEO correctly identified data flows to privileged operations, these operations were actually protected by authN/authZ mechanisms that NEO did not locate. For example, checks implemented through external services or configuration files were not visible in NEO's analysis.

False Negatives. We analyzed the 3 false negatives in the ground-truth dataset to understand NEO's limitations. One case was caused by NEO failing to detect the privileged operation that was invoked through uncommon API patterns. The second case was caused by a missed flow, even though the privileged operation was present. The third case reveals a limitation in handling framework-specific URL processing semantics. Specifically, CVE-2023-38493 [62] exploited matrix variables—a URL feature that allows embedding key-value pairs within path segments using semicolons (*e.g.*, `/path;key=value/resource`). An attacker could bypass authentication by sending `/important;a=b/resources` to access a protected endpoint configured with `decoratorUnder("/important/", authService)`. While NEO correctly identified the authentication decorator as a security check, it failed to recognize that Spring's routing logic would normalize the matrix variable path to `/important/resources`, causing the request to bypass the decorator's path matching. This occurred because NEO failed to recognize this framework-specific URL parsing that could alter access control decisions at runtime.

Summary: NEO detected 24 new zero-day privilege escalation vulnerabilities in real-world microservices, and achieved 85.0% recall and 81.0% precision on the ground-truth dataset.

6.3. Ablation Study

We conduct an ablation study to evaluate the performance of individual components in NEO and how they contribute to the overall performance. We also characterize different LLM variations.

TABLE 5: Detection effectiveness in the ablation study. Recall is computed with respect to the total of 42 vulnerabilities (24 new and 18 known) described in §6.2.

Variants	TP	FP	FN	Prec.	Recall
$\text{NEO}_{basic-sink}$	14	0	28	100.0%	33.3%
$\text{NEO}_{no-queryop}$	2	0	40	100.0%	4.8%
$\text{NEO}_{no-odctx}$	27	31	15	46.6%	64.3%
NEO	39	12	3	76.5%	92.9%

6.3.1. Detection Effectiveness. We design the following variants of NEO to understand the vulnerability detection effectiveness of NEO's components. We evaluate these variants on 42 vulnerabilities described in §6.2, including 22 vulnerabilities from the evaluation corpus and 20 vulnerabilities from the ground-truth dataset.

NEO-basic-sink. A major component of NEO is identifying privileged operations beyond common patterns. We thus design NEO-basic-sink to use only a list of standard privileged operations (*e.g.*, networking, database, and file access) as adopted in prior work [15], [20] without application-specific privileged operations generated by LLMs.

NEO-no-queryop. NEO employs code search primitives for flexible code search. In NEO-no-queryop, we remove these query operations. Instead, we require NEO-no-queryop to directly generate CodeQL queries for code search.

NEO-no-odctx. On-demand context retrieval is another key technique in NEO. To understand its necessity, NEO-no-odctx allows only a one-time context retrieval after the cross-service flow is produced, instead of doing it iteratively.

Results. Table 5 presents the results of our ablation study on the 42 vulnerabilities mentioned in §6.2. Again, the full-fledged NEO system correctly identified 39 out of 42 vulnerabilities with only 3 false negatives. In contrast, all three ablated variants show significantly degraded performance, demonstrating that each component plays a critical role in NEO's effectiveness.

NEO-basic-sink with only standard privileged operations identified 14 vulnerabilities (missed 28 vulnerabilities) with no false positives, achieving substantially lower recall of 33.3%. This result demonstrates that basic operations are insufficient for comprehensive vulnerability detection and cannot cover many privilege escalation vulnerabilities involving application-specific privileged operations. This is

also consistent with our characterizations of true positives mentioned in §6.2.

Removing the code search primitives in NEO-no-queryop resulted in the most dramatic performance degradation. This variant detected only 2 vulnerabilities while missing 40, with a recall dropping to 4.8%. The results indicate that direct CodeQL query generation is extremely challenging, and LLMs mostly generate invalid CodeQL queries with a lot of syntax errors. The code search primitives provide essential abstractions that enable NEO to systematically search for security-relevant code patterns.

NEO-no-odctx with only a one-time context retrieval identified 27 vulnerabilities, but produced 31 false positives and missed 15 vulnerabilities. The substantial decline in both precision and recall underscores the importance of iterative context retrieval for handling complex vulnerability cases. Without the ability to dynamically gather and integrate additional context during analysis, NEO struggles to locate authN/authZ checks, leading to both missed detections and incorrect assessments.

6.3.2. Code Search Primitives. Beyond vulnerability detection, we evaluate the correctness of the search results produced by the code search primitives. Specifically, we aim to determine whether search execution yields false positives (*i.e.*, incorrect results) or false negatives (*i.e.*, missing results). However, such evaluation is challenging, as it requires constructing a complete ground truth for the expected query results across all application code. One potential approach is to use dynamic analysis to collect execution traces, but this method suffers from significant coverage limitations [63]. To bridge this gap, we conducted a rigorous manual assessment on a single application, PiggyMetrics [37], rather than on the entire dataset. We first recorded the complete trajectory of 132 unique executed code search primitives and their corresponding results during the analysis. From this set, we performed stratified random sampling, selecting 5 instances for each primitive type (Q_{name}, Q_{ast}, Q_{flow}, and Q_{cg}) to ensure a representative distribution across different search behaviors. This resulted in a total of 20 searches for manual audit. To establish a reliable ground truth, we exhaustively analyzed the target codebase for each search to verify the accuracy of the returned results and pinpoint the exact nature of any observed failures.

Imprecise Search Results. Among the 20 sampled searches, we observed imprecise results in 7 instances, particularly within the types Q_{flow} and Q_{cg}. These searches are not entirely incorrect but often return over-approximated results due to the inherent limitations of static analysis. For example, due to over-approximation in virtual dispatch, CodeQL's `polyCalls` (and similar predicates) frequently included edges to all potential method implementations because it could not statically determine the specific runtime target. These issues are potentially fixable by refining the underlying predicates in CodeQL, which we leave for future work. We encountered no imprecision in Q_{name} and Q_{ast}.

Missing Search Results. Among the 20 sampled searches, we observed that results were missing in a total of 5 instances, specifically within the Q_{flow} and Q_{cg} categories. These searches often return partial results due to the conservative nature of static analysis when handling complex control flow or implicit dependencies. Similar to the imprecise results, these omissions highlight the challenges of achieving perfect recall in automated code analysis.

6.3.3. Validation Strategies. To understand how the two validation strategies help reduce false positives, we present the statistics in Figure 6. In our dataset, the cross-service flow analysis initially produced 135 flows from user inputs to privileged operations. Among them, NEO generated constraints for 28 cases, all of which were in syntactically valid SMT-LIB format. 15 flows (11.1%) were filtered out by proving through constraint solving that their corresponding constraints are infeasible. We manually confirmed that all 15 cases were true negatives without any false negatives introduced.

Subsequently, the LLM-based authN/authZ validation examined the remaining flows and eliminated 69 that lack proper authN/authZ checks but are semantically benign based on code context. Together, these two strategies reduced the initial 135 flows to 51 final reports, eliminating 84 potential false positives (a 62.2% reduction). This analysis highlights that both validation strategies help eliminate false positives.

To assess the precision of the LLM-based authN/authZ validation component, we manually audited 10 randomly sampled cases from the 69 flows. By cross-referencing the LLM's justifications with the source code, we confirmed that in all 10 instances, the LLM correctly identified the explicit decorators (*e.g.*, `@RequiresAdmin()`) or implicit logic that protects the privileged operations. This suggests that this validation strategy successfully reduces false positives without introducing false negatives in our samples. However, we acknowledge that this LLM-based reasoning is inherently probabilistic and may occasionally yield incorrect classifications. To mitigate this risk, one could adopt a human-in-the-loop approach [64], allowing security engineers to check the LLM's justifications.

6.3.4. LLM Variations. NEO relies on LLMs for core reasoning tasks. In this section, we analyze how performance is influenced by model selection, model temperature (stochasticity), and prompting strategy. Due to the high cost of model APIs (§6.5), running this sensitivity analysis on the complete dataset is computationally and financially prohibitive. Therefore, we evaluate this sensitivity analysis on the ground-truth dataset used in §6.2, which includes the previously reported vulnerabilities and the additional vulnerabilities uncovered by NEO in those same applications (Table 4). Accordingly, recall in this subsection is measured over the ground-truth dataset.

Models and Temperatures. We evaluated NEO with three different models: (1) Claude 3.7 Sonnet, which is the default model throughout our experiments, (2) Gemini 2.5 Flash from Google, and (3) GPT-5 mini from OpenAI. We analyzed the effect of stochasticity by testing temperatures $T \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$.

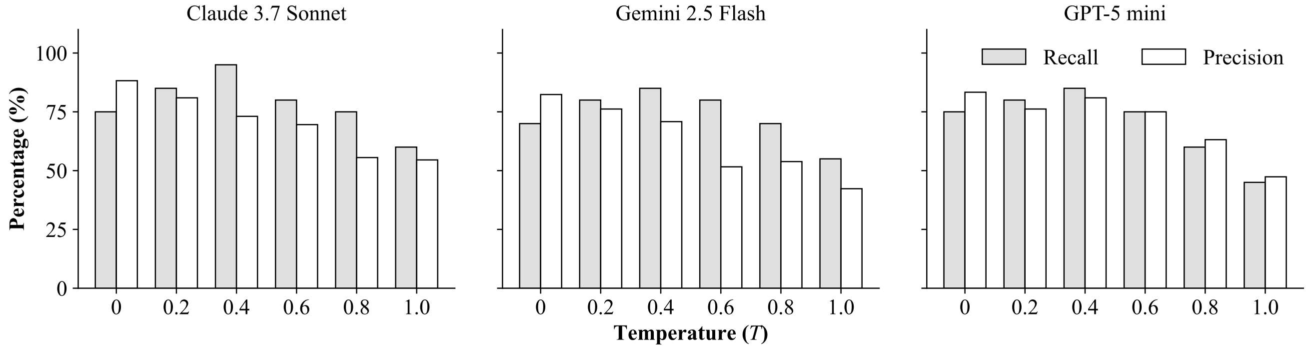


Figure 5: Impact of model selection and temperature on vulnerability detection.

As shown in Figure 5, the performance of NEO is highly sensitive to both model selection and temperature (T). Lower temperatures (*e.g.*, $T \leq 0.2$) maximize precision; notably, GPT-5 mini yields the fewest false positives by strictly adhering to known security patterns. However, these deterministic settings occasionally miss complex, multi-step vulnerabilities. In contrast, mid-range temperatures (*e.g.*, $T \in [0.4, 0.6]$) improve recall by enabling exploratory reasoning, allowing Claude 3.7 Sonnet to uncover additional logical flaws at $T = 0.4$ that were missed at $T = 0$. Beyond $T = 0.8$, performance degrades across all models due to increased hallucinations and the flagging of benign code.

Prompting Strategies. By default, the NEO prompt includes two in-context examples (few-shot) and a structured three-step workflow to facilitate Chain-of-Thought (CoT) reasoning. To isolate the impact of these components, we evaluate two additional strategies. First, we test *zero-shot prompting*, where we remove the two demonstration examples to assess the model's reliance on in-context learning and its ability to identify vulnerabilities from its pre-training data. Second, we evaluate *direct prompting (No-CoT)*, where we remove the three-step reasoning instructions and require the model to output the final results directly. The same in-context examples are used.

TABLE 6: Ablation of prompting strategies on Claude 3.7 Sonnet ($T = 0.2$), evaluated on the ground-truth dataset.

Strategy	TP	FP	Prec.	Recall
NEO (Few-shot + CoT)	17	4	81.0%	85.0%
Zero-shot	14	6	70.0%	70.0%
Direct (No-CoT)	11	9	55.0%	55.0%

As shown in Table 6, the default NEO configuration achieves the highest performance, recovering 17 of the 20 vulnerabilities (85.0% recall) on this ground-truth dataset. Removing few-shot demonstrations reduces the model's ability to ground its analysis, leading to a drop in both precision and recall. Furthermore, removing the CoT reasoning workflow results in the most significant degradation, confirming that intermediate Chain-of-Thought steps are essential for tracing complex data flows and minimizing false positives.

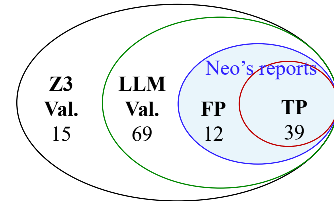


Figure 6: Impact of validation strategies. There are 135 flows initially.

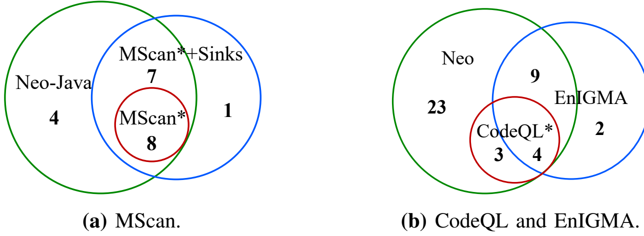
Summary: Application-specific privileged operations increase recall from 33.3% to 92.9%, while code search primitives enable flexible search and avoid failures. The validation strategies together eliminate 62.2% of false positives.

6.4. Comparison

In this section, we compare NEO with prior classic and agentic state-of-the-art program analysis approaches on our dataset. Ideally, a comparison would require labeling all vulnerabilities in the evaluated applications with ground truth. However, this would require significant manual effort and is challenging (if not infeasible) to ensure completeness. Therefore, we follow a common practice [20], [65] and construct a consolidated comparison set as the union of vulnerabilities identified by all evaluated baselines and NEO. In total, this *consolidated comparison set* consists of 44 vulnerabilities, including 42 vulnerabilities described in §6.2 and 2 additional vulnerabilities identified by a baseline [26].

6.4.1. Baselines. We compared NEO against three state-of-the-art approaches: CodeQL [23], MScan [20], and EnIGMA [26].

MScan. MScan [20] is a state-of-the-art static data-flow analysis tool designed to detect taint-style vulnerabilities in Java-based microservices. While MScan was not originally designed for privilege escalation detection, it represents the closest comparable approach in the microservice security domain, as it performs taint analysis to identify flows to security-critical operations and supports cross-service



(a) MScan. **(b)** CodeQL and EnIGMA.

Figure 7: Distribution of true positive detections by each approach. MScan and CodeQL are integrated with Neo's validation and marked with a suffix *. MScan* is evaluated on Java-only vulnerabilities, while all other approaches are evaluated on the full polyglot dataset. MScan*+Sinks denotes MScan* enhanced with Neo's identified privileged operations.

interaction analysis. To enable a meaningful comparison, we integrate Neo's validation components to adapt MScan for privilege escalation detection, denoted as MScan*. The key difference between vanilla MScan and MScan* is that the latter incorporates Neo's permission check validation, while vanilla MScan only performs taint flow analysis without validating whether appropriate authorization checks exist. We further build MScan*+Sinks, which extends MScan* with Neo's identified application-specific privileged operations, replacing MScan's default set of common security-critical operations. Due to MScan's language limitation, we evaluate it only on the Java-based microservices in our dataset. To ensure a rigorous and fair comparison, we applied Neo 's post-processing validation to the raw outputs of MScan* and MScan*+Sinks. This filters common static analysis noise, allowing for a direct comparison of each tool's capability against Neo.

CodeQL. CodeQL [23] is a widely used static analysis tool that we include as a baseline for vulnerability detection. It supports all programming languages in our dataset through its multi-language analysis framework. However, CodeQL lacks inter-service analysis capabilities and treats each service independently, making it unable to detect cross-service privilege escalation vulnerabilities spanning multiple microservices. We run CodeQL's default security query suite in the critical category and apply it to each service individually. Similarly, we integrate Neo's validation components with CodeQL's results, denoted as CodeQL*.

EnIGMA. EnIGMA [26] is an LLM-powered security testing agent developed based on SWE-agent [24], which leverages LLMs to automatically discover vulnerabilities through dynamic code exploration and reasoning. As introduced in §2.3, EnIGMA relies on basic bash commands (*e.g.*, `grep`, `find`) to navigate and read code files. It loads code into the LLM's context window to reason about data flows and security implications directly through natural language understanding. For meticulous whole-project analysis, we use the same setup as Neo. Specifically, we provide the same standardized prompt strategy used by Neo (§5), together with the application's documentation. The same time budget of 10 hours is enforced for running EnIGMA.

TABLE 7: TP, FP, and precision of compared tools.

	MScan*	MScan*+Sinks	CodeQL*	EnIGMA	Neo
TP	8	16	7	15	39
FP	3	7	6	14	12
Prec.	72.7%	69.6%	53.8%	51.7%	76.5%

6.4.2. MScan Comparison. Overall, MScan* found 8 privilege escalation vulnerabilities while MScan*+Sinks found 16, with the additional 8 vulnerabilities benefiting from the application-specific privileged operations identified by Neo. In comparison, Neo found 19 vulnerabilities on the same Java-based microservices. We present the distribution of true positives found by each tool in the Venn diagram in Figure 7(a). The majority of vulnerabilities detected by MScan* or MScan*+Sinks were also found by Neo. The only exception is CVE-2023-38493 [62], which was detected by MScan*+Sinks but not by Neo, as explained earlier in §6.2. This occurred because MScan*+Sinks' (or MScan*'s) path analysis considered Spring's matrix variable normalization behavior, whereas Neo did not model this framework-specific URL processing semantic. MScan* and MScan*+Sinks reported 3 and 7 false positives, leading to a precision of 72.7% and 69.6%, respectively.

6.4.3. CodeQL Comparison. CodeQL* detected 7 privilege escalation vulnerabilities, significantly fewer than Neo's 39 total detections across all languages. As shown in Figure 7(b), all vulnerabilities found by CodeQL* were also covered by Neo. This demonstrates the superior detection capability of Neo, which benefits from cross-service analysis and application-specific privileged operation detection. CodeQL* achieved a precision of 53.8% with 6 false positives.

6.4.4. EnIGMA Comparison. EnIGMA detected 15 privilege escalation vulnerabilities out of the 44 in the consolidated comparison set with 14 false positives. As shown in Figure 7(b), Neo detected 26 vulnerabilities that EnIGMA missed, demonstrating superior detection coverage. EnIGMA 's failures are primarily due to context exhaustion and the inefficiency of bash-based exploration. For example, in DeathStarBench [54], a single data-flow path could span over 35 functions across different services. Resolving the flows via bash commands returned massive irrelevant code that overwhelms EnIGMA to eventually give up the analysis. Neo succeeded because search primitives could directly retrieve context relevant to the vulnerability path. This demonstrates Neo's advantage in static guidance, which filters out noise across large codebases.

However, EnIGMA found 2 vulnerabilities that Neo missed. We analyzed EnIGMA's execution trajectories when interacting with the LLM to understand how it discovered these vulnerabilities and why Neo missed them. In both cases, Neo successfully identified the privileged operations but failed to construct complete data flows due to complex dynamic function calls in JavaScript that its flow query operations could not track. Consequently, these flows never reached Neo's LLM component for further analysis. In

TABLE 8: New vulnerabilities found in our extensibility study.

Vulnerability Type	TP	Ack'ed	Fixed
Privilege Escalation	11	5	2
Command Injection	5	2	0
SQL Injection	2	0	0

contrast, EnIGMA directly reads and reasons about code through natural language understanding, without relying on program analysis to construct data flows. This allows EnIGMA a chance to identify flows obscured by dynamic language features.

Summary: NEO detected 39 of 44 vulnerabilities, substantially outperforming MScan, EnIGMA, and CodeQL. This can be attributed to its LLM-based privileged operation identification and code search primitives.

6.5. Efficiency and Cost

We measured the analysis time and API cost of NEO when conducting the experiments. NEO took a total of 38.9 hours to complete the analysis of all 25 applications, averaging 1.6 hours per application. This end-to-end time includes privileged operation identification, cross-service flow analysis, on-demand context retrieval, LLM invocations, and final validation. The average API cost was approximately 18 USD per application. Given the complexity and scale of the microservices in our dataset, the efficiency and cost are practical for real-world deployment. As a comparison, on average, EnIGMA took 3.4 hours and around 25 USD per application. Note that the API cost of EnIGMA varied significantly across applications. It frequently triggered early termination or, conversely, led to repeated context exhaustion.

6.6. Extensibility

The underlying design of NEO is not limited to privilege escalation detection in microservices. We argue that NEO has broad applicability across different contexts with only minor modifications. To validate this claim, we apply NEO to detect privilege escalation in generic applications and to detect other types of vulnerabilities.

Adaptability to Generic Applications. By design, the code search primitives and agentic program analysis pipeline in NEO are not specific to microservices. The core techniques, such as code search and flow analysis, remain applicable to generic applications. To validate how NEO adapts to generic applications, we modified the prompts to specify appropriate program context rather than assuming microservice architecture, and applied the modified NEO to several real-world open-source Python and Java applications. As shown in Table 8, the modified NEO successfully identified 11 new privilege escalation vulnerabilities in these generic applications, with 5 acknowledged and 2 fixed by developers. These results demonstrate that NEO's techniques generalize beyond microservices, confirming that the code search primitives and agentic analysis pipeline are broadly applicable.

Extensibility to Other Vulnerability Types. While NEO includes specialized components for privilege escalation (*e.g.*, privileged operation identification and authN/authZ detection), its core architecture generalizes to any taint-style, data-flow-based vulnerability. The source-to-sink analysis paradigm naturally applies to SQL injection, XSS, command injection, and similar vulnerability classes that are prevalent in web and cloud applications. Adapting NEO requires only redefining sources, sinks, and sanitizers through prompts, and the analysis pipeline remains unchanged.

We configured NEO to detect injection-based vulnerabilities [65] via prompt modifications. As shown in Table 8, NEO identified 5 command injection and 2 SQL injection vulnerabilities in JavaScript-based web applications and packages, with 2 acknowledged by developers. This confirms NEO's extensibility to diverse vulnerability types without architectural changes.

6.7. Case Study

We present a case study to illustrate how NEO identified a privilege escalation vulnerability in Newbee Mall [58], [66], a popular e-commerce application with 11.4K stars on GitHub. As shown in Figure 8, the vulnerability exists in a payment endpoint where the `paySuccess` function allows users to mark orders as paid by providing an order number. While it validates that the order is in prepaid status (`ORDER_PRE_PAY`), it lacks authorization to verify order ownership. Any authenticated user can thus mark arbitrary orders as paid without actual payment.

NEO identifies this vulnerability through its multi-stage pipeline. First, NEO identifies payment update operations as privileged by searching code with the function name `order` using Q_{ast} and Q_{name}, and the returned function calls are then validated by the LLM, which identifies lines 11-13 as privileged operations. Second, code search primitives trace the flow from user-controlled endpoints to the identified privileged operations. Third, the LLM-based validator explores the code to detect possible authN/authZ checks and identifies the missing ownership check, then further assesses its security impact. During this process, on-demand context retrieval examines other similar methods in the file, such as `cancelOrder()`, and discovers that they properly implement ownership verification (*e.g.*, `order.getUserId().equals(currentUser.getId())`), confirming this is a genuine vulnerability.

6.8. Responsible Disclosure

We follow standard responsible disclosure practices to protect users and developers from potential harm. For each discovered vulnerability, we notified the corresponding developers or project maintainers. As of this submission, 18 out of 42 new vulnerabilities from §6.2 and §6.6 have been acknowledged by developers, and 8 have been patched. We continue to monitor the status of the remaining reports and communicate with developers to ensure timely fixes.

```java
@GetMapping("/paySuccess")
public String paySuccess(String orderNo, int payType) {
  NbOrder order = nbOrderMapper.selectByOrderNo(orderNo);

  if (order != null) {
    // Only allow payment for orders in pre-pay status
    if (order.getOrderStatus().intValue() != ORDER_PRE_PAY) {
      return ResultEnum.ORDER_STATUS_ERROR.getResult();
    }
    // change order to paid
    order.setOrderStatus((byte) ORDER_PAID);
    order.setPayType((byte) payType);
    order.setPayStatus((byte) PAY_SUCCESS);
    ...
    }
}
```

Figure 8: A (simplified) privilege escalation in Newbee Mall that allows setting an arbitrary order as paid.

7. Discussion

Limitations. We acknowledge a few limitations of NEO's current implementation. First, NEO's code search primitives are built on CodeQL and inherit its limitations in handling dynamic language features (*e.g.*, dynamic function calls and reflection) and pointer aliases. While NEO supports multiple programming languages, its effectiveness is also restricted by CodeQL's language support and the quality of language-specific models. One can also prototype the idea of code search primitives using other analysis backends such as Joern [31]. Second, NEO's code analysis component focuses on source code and cannot directly process configuration files (*e.g.*, YAML, JSON, XML) that may contain relevant security settings such as access control policies or service routing rules. Therefore, NEO relies on LLMs to interpret configuration content when retrieved as context, rather than performing structured semantic analysis. Vulnerabilities stemming from such misconfigurations may be missed if the relevant configuration files are not retrieved into context. Furthermore, many misconfigurations often depend on real-world deployment environments, which are typically inaccessible in a third-party red-teaming scenario. Given these constraints, privilege escalation via misconfiguration is out of scope for NEO. Interested readers may refer to dedicated tools like ConfigX [67] for detecting them.

Future Work. Beyond addressing the aforementioned limitations, several promising directions could extend NEO's capabilities and impact. First, to mitigate the false positives, such as the ones identified in §6.2, we plan to refine the LLM's context-awareness to better distinguish execution environments, such as identifying logic unique to client-side components. Furthermore, we intend to explore hybrid analysis approaches that combine static analysis with dynamic testing and under-constrained symbolic execution. This could enable NEO to navigate complex authN/authZ mechanisms and facilitate the automatic generation of proof-of-concept exploits to verify vulnerabilities. Finally, we plan to investigate how incremental analysis techniques could enable NEO to efficiently analyze code changes in continuous integration pipelines, making it more practical for ongoing development workflows.

8. Related Work

Program Analysis for Microservices. MScan [20] performs cross-service data-flow analysis for taint-style vulnerabilities in Java-based microservice applications. It does not model or consider permissions or access control for privilege escalations. ANTaint [68] builds an efficient and sound call graph and handles taint propagation in libraries by specifying shortcut rules over function arguments and return values. Although both techniques target Java applications, their ideas are complementary to NEO. For example, NEO currently builds on CodeQL for multi-language support, which also constrains it by CodeQL's precision in call-target resolution and the resulting call graph quality. Microscope [21] performs change-impact analysis to identify how code changes in one service propagate to others. It models polyglot microservices using a unified Datalog representation and defines Datalog rules to determine the set of impacted public interfaces. In contrast, NEO conducts a more fine-grained cross-service data-flow analysis that explicitly tracks the propagation path from user inputs to privileged operations, enabling the detection of security vulnerabilities. AUTOARMOR [11] statically analyzes microservice source code to extract inter-service access control policies that define which services are authorized to make network requests to others.

Privilege Escalation Detection. An open challenge in this domain is inferring the intended security policy for privileged operations. Prior code analysis work often employs heuristics to approximate these intended policies [15]–[19]. MACE [15] computes authorization contexts (comprising users, roles, session variables, and permissions associated with sensitive operations) and compares these contexts across related operations (*e.g.*, database `INSERT` and `DELETE`). It flags inconsistencies as potential cases of missing authorization checks. RoleCast [16] further utilizes knowledge of user roles and their associated permissions to detect inconsistencies. Another line of work focuses on identifying existing permission checks and flagging cases where such checks are missing. PCFinder [17], for instance, leverages naming conventions of credentials (*e.g.*, variables such as `password`) and other semantic cues to detect permission checks. MOCGuard [18] analyzes database operations to infer data ownership and identifies missing owner-checks. MPChecker [19] refers to the application log mechanisms to infer privileged operations that necessitate permission checks. NEO approaches the open challenge in the context of polyglot microservices by leveraging LLMs for policy inference.

LLM-based Program Analysis. Leveraging LLMs for program analysis has become an emerging trend. RepoAudit [69] employs LLMs to navigate large codebases and perform on-demand analyses. Recent research also explores agentic and hybrid techniques that combine LLM reasoning with traditional static or dynamic analyses. IRIS [70], for instance, uses LLMs to infer taint sources and sinks and augments existing CodeQL queries with the inferred information. LLMxCPG [71] constructs program slices and leverages an LLM to validate them, while LLMSA [34] decomposes analysis tasks into smaller property checks for improved

accuracy and interpretability. LLift [30] applies LLM reasoning to infer post-conditions for use-before-initialization bugs in the Linux kernel. In contrast, NEO emphasizes the importance and necessity of rigorous program analysis to support scalable and efficient analysis, which could not be handled with only LLMs.

9. Conclusion

In this work, we presented NEO, an LLM-based agentic program analysis framework for detecting privilege escalation vulnerabilities in polyglot microservices. NEO leverages LLMs to reason about natural language semantics and employs rigorous program analysis to analyze structured code. Our carefully designed code search primitives enable NEO to flexibly and scalably explore large codebases. On real-world microservice applications, our evaluation uncovered 24 zero-day privilege escalation vulnerabilities. NEO significantly outperformed existing approaches in detection accuracy, scalability, and extensibility. We believe such an agentic program analysis pipeline represents a promising paradigm for addressing emerging software security challenges in the long term.

Ethics Considerations

We identify application owners and their end users as the primary stakeholders that may potentially be affected by our work. To minimize potential risks and harms, we adopted a proactive, responsible disclosure approach by reporting newly identified vulnerabilities immediately upon discovery. Our evaluation spanned two months, from September to October 2025, during which we reported findings to affected software vendors and strongly encouraged prompt remediation. To prevent in-the-wild exploitation, we did not publicly disclose vulnerability details for any reported issues prior to vendor patches being made available. As of this submission, 18 out of 42 vulnerabilities have been acknowledged by developers, and 8 have been patched.

All experiments, vulnerability validation, and proof-of-concept testing were conducted exclusively in isolated local environments under our control. We reconstructed vulnerable microservice configurations locally to reproduce and verify privilege escalation paths without interacting with production systems or accessing real user data. Our methodology ensured no live deployments were compromised and no real-world service disruptions were caused.

NEO employed LLMs to analyze application code and identify privilege escalation vulnerabilities. To ensure responsible use, all LLM analysis was conducted in isolated environments and excluded sensitive credentials, proprietary code, and personally identifiable information. Final LLM outputs were manually validated by security researchers before reporting to vendors.

We acknowledge that NEO, as a vulnerability detection framework, could potentially be extended or misused for malicious purposes. To mitigate this risk, we release our artifact in a research-oriented form, excluding specific vulnerability details and exploit payloads. We believe the societal benefits of this work significantly outweigh the limited risks.

LLM Usage Considerations

LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality. The literature review, NEO framework design, system implementation, and experimental evaluation were all conducted independently by the authors without LLM assistance. During the writing process, we employed LLMs to polish the manuscript by improving sentence structure, correcting grammar, and enhancing the clarity of technical explanations. All substantive content, including the methodological approach, experimental findings, security analysis, and research conclusions, represents the authors' original contributions. We carefully reviewed all LLM-generated editorial suggestions to verify their accuracy and appropriateness before incorporation.

LLMs are used in NEO's design for vulnerability detection. NEO leverages Claude Sonnet 3.7 as an agentic reasoning component that dynamically generates analysis plans, adapts code search strategies, and validates security semantics. All LLM-generated analysis decisions were executed in controlled environments for analyzing open-source microservice applications. Our use of a closed-source model may introduce reproducibility challenges. To mitigate that, we document exact model versions and parameters, and provide major prompts used in the framework in the artifact. We manually verified all detection results to confirm their validity and responsibly disclosed the vulnerabilities to the relevant developers for remediation. We present all detection statistics in the paper.

We do not train any models in this work. NEO analyzes open-source microservice applications from publicly available GitHub repositories, which do not raise concerns regarding consent, data holder rights, or intellectual property. Our environmental footprint is limited to inference costs totaling 38.9 hours of LLM API usage, as discussed in Section 6.5. This cost is justified by NEO's goal of automating privilege escalation detection in complex polyglot microservices, a task requiring extensive manual security expertise. We minimized our footprint by using code search primitives to retrieve only relevant code snippets instead of reading entire codebases.

Acknowledgments

The authors would like to thank the anonymous reviewers and shepherd for their constructive comments. We also thank Andrew Johnston for the valuable discussions. This work was supported in part by the National Science Foundation (NSF) under grants CNS-21-54404 and CNS-20-46361, Gifts from Google, the Columbia Research Stabilization Fund, and the Columbia OPA Academic Conference Travel Grant. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily

representing the official policies or endorsements, either expressed or implied, of the funding agencies.

References

[1] C. Richardson, "Microservice architecture," 2025, https://microservices.io/.

[2] S. Newman, *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media, 2019.

[3] Y. Abgaz, A. McCarren, P. Elger, D. Solan, N. Lapuz, M. Bivol, G. Jackson, M. Yilmaz, J. Buckley, and P. Clarke, "Decomposition of monolith applications into microservices architectures: A systematic review," *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4213–4242, 2023.

[4] R. Brigham, "What led amazon to its own microservices architecture," 2015, https://thenewstack.io/led-amazon-microservices-architecture/.

[5] R. Shoup, "Microservice ecosystems of google and ebay," 2015, https://highscalability.com/deep-lessons-from-google-and-ebay-on-building-ecosystems-of/.

[6] A. W. Services, "What are microservices?" 2025, https://aws.amazon.com/microservices/.

[7] G. Cloud, "Microservices architecture on google cloud," 2025, https://cloud.google.com/blog/topics/developers-practitioners/microservices-architecture-google-cloud.

[8] Gz-yami, "Mall4Cloud: A microservices-based e-commerce platform," 2025, accessed: October 2025. [Online]. Available: https://github.com/gz-yami/mall4cloud

[9] "Crayfish homarus microservice remote code execution vulnerability," 2024, https://nvd.nist.gov/vuln/detail/CVE-2025-25286.

[10] "Cisco ultra cloud core - subscriber microservices infrastructure privilege escalation vulnerability," 2022, https://www.cisco.com/c/en/us/support/docs/csa/cisco-sa-uccsmi-prvesc-BQHGe4cm.html.

[11] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen, "Automatic policy generation for $inter-service$ access control of microservices," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3971–3988.

[12] "grpc: A high performance, open source universal rpc framework," 2025, https://grpc.io/.

[13] R. T. Fielding, "Representational state transfer (rest)," 2000, https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[14] "Apache kafka: A distributed streaming platform," 2025, https://kafka.apache.org/.

[15] M. Monshizadeh, P. Naldurg, and V. Venkatakrishnan, "Mace: Detecting privilege escalation vulnerabilities in web applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 690–701.

[16] S. Son, K. S. McKinley, and V. Shmatikov, "Rolecast: finding missing security checks when you do not know what checks are," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, 2011, pp. 1069–1084.

[17] Y. Shi, F. Liu, G. Yang, Y. Zhang, Y. Cao, E. Li, X. Tan, X. Luo, M. Yang, and S. Chen, "Facilitating access control vulnerability detection in modern java web applications with accurate permission check identification," *IEEE Transactions on Information Forensics and Security*, 2025.

[18] F. Liu, Y. Shi, Y. Zhang, G. Yang, E. Li, and M. Yang, "Mocguard: Automatically detecting missing-owner-check vulnerabilities in java web applications," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 903–919.

[19] 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*, 2022, pp. 2145–2158.

[20] F. Liu, Y. Zhang, T. Chen, Y. Shi, G. Yang, Z. Lin, M. Yang, J. He, and Q. Li, "Detecting taint-style vulnerabilities in microservice-structured web applications," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 972–990.

[21] Q. Shi, X. Xie, X. Fu, P. Di, H. Li, A. Zhou, and G. Fan, "Datalog-based language-agnostic change impact analysis for microservices," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025, pp. 652–652.

[22] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *international conference on Information security*. Springer, 2010, pp. 346–360.

[23] GitHub, Inc., "CodeQL: Discover vulnerabilities across a codebase with queries," 2025, accessed: October 2025. [Online]. Available: https://codeql.github.com/

[24] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. [Online]. Available: https://arxiv.org/abs/2405.15793

[25] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig, "Openhands: An open platform for AI software developers as generalist agents," in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: https://openreview.net/forum?id=OJd3ayDDoF

[26] T. Abramovich, M. Udeshi, M. Shao, K. Lieret, H. Xi, K. Milner, S. Jancheska, J. Yang, C. E. Jimenez, F. Khorrami, P. Krishnamurthy, B. Dolan-Gavitt, M. Shafique, K. R. Narasimhan, R. Karri, and O. Press, "EnIGMA: Interactive tools substantially assist LM agents in finding security vulnerabilities," in *Forty-second International Conference on Machine Learning*, 2025. [Online]. Available: https://openreview.net/forum?id=Of3wZhVv1R

[27] GraphQL Foundation, "GraphQL specification," 2021, accessed: October 2025. [Online]. Available: https://spec.graphql.org/September2025/

[28] VMware, Inc., "RabbitMQ: Messaging that just works," 2024, accessed: October 2025. [Online]. Available: https://www.rabbitmq.com/

[29] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Demystifying llm-based software engineering agents," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3715754

[30] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3649828

[31] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 590–604.

[32] GitHub, Inc., "CodeQL queries," 2025, accessed: October 2025. [Online]. Available: https://codeql.github.com/docs/writing-codeql-queries/codeql-queries/

[33] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[34] C. Wang, Y. Gao, W. Zhang, X. Liu, Q. Shi, and X. Zhang, "Llmsa: A compositional neuro-symbolic approach to compilation-free and customizable static analysis," in *Proceedings of the 2024 Empirical Methods in Natural Language Processing (EMNLP)*, Miami, FL, USA, Nov. 2024.

[35] Z. Luo, H. Zhao, D. Wolff, C. Cadar, and A. Roychoudhury, "Agentic concolic execution," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2026.

[36] Zealon159, "Light Reading Cloud: A microservices-based reading application," 2025, accessed: October 2025. [Online]. Available: https://github.com/Zealon159/light-reading-cloud

[37] sqshq, "PiggyMetrics: Microservice architecture with spring boot, spring cloud and docker," 2025, accessed: October 2025. [Online]. Available: https://github.com/sqshq/piggymetrics

[38] EdwinVW, "Pitstop: A sample application based on a garage management system for pitstop - a fictitious garage," 2025, accessed: October 2025. [Online]. Available: https://github.com/EdwinVW/pitstop

[39] sitewhere, "SiteWhere: An industrial strength open-source application enablement platform for the internet of things (iot)," 2025, accessed: October 2025. [Online]. Available: https://github.com/sitewhere/sitewhere

[40] ZongXR, "SuperMarket: A project that simulates a real-world supermarket or retail store billing experience," 2025, accessed: October 2025. [Online]. Available: https://github.com/ZongXR/SuperMarket

[41] mehdihadeli, "Food Delivery (.NET): A practical food delivery microservices, built with .net 8, masstransit, domain-driven design, cqrs, and more," 2025, accessed: October 2025. [Online]. Available: https://github.com/mehdihadeli/food-delivery-microservices

[42] GoogleCloudPlatform, "Online Boutique: Sample cloud-first application with 10 microservices showcasing kubernetes, istio, and grpc," 2025, accessed: October 2025. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

[43] meysamhadeli, "Booking Microservices: A practical microservices with the latest technologies and architectures like vertical slice architecture, event sourcing, cqrs, ddd, grpc, and .net 9," 2025, accessed: October 2025. [Online]. Available: https://github.com/meysamhadeli/booking-microservices

[44] harlow, "Go Micro Services: An example of microservices in go using grpc," 2025, accessed: October 2025. [Online]. Available: https://github.com/harlow/go-micro-services

[45] 417511458, "JBone: A microservice platform based on spring cloud," 2025, accessed: October 2025. [Online]. Available: https://github.com/417511458/jbone

[46] FudanSELab, "Train Ticket: A benchmark microservice system," 2025, accessed: October 2025. [Online]. Available: https://github.com/FudanSELab/train-ticket

[47] aspnetrun, "AspnetRun E-Shop: Microservices on .net platforms used asp.net web api, docker, rabbitmq, masstransit, grpc, yarp api gateway, and more," 2025, accessed: October 2025. [Online]. Available: https://github.com/aspnetrun/run-aspnetcore-microservices

[48] crizstian, "Cinema Microservice: A nodejs microservice for a cinema booking system," 2025, accessed: October 2025. [Online]. Available: https://github.com/crizstian/cinema-microservice

[49] elgris, "TODO App: An example microservice app written in different languages (go, java, nodejs, python, vuejs)," 2025, accessed: October 2025. [Online]. Available: https://github.com/elgris/microservice-app-example

[50] abpframework, "eShopOnAbp: Reference microservice solution built with the abp framework and .net," 2025, accessed: October 2025. [Online]. Available: https://github.com/abpframework/eShopOnAbp

[51] anilallewar, "Spring Boot Basics Demo: Basic architecture framework to create complete microservices using spring boot and spring cloud," 2025, accessed: October 2025. [Online]. Available: https://github.com/anilallewar/microservices-basics-spring-boot

[52] magda-io, "Magda: A federated, open-source data catalog for all your big data and small data," 2025, accessed: October 2025. [Online]. Available: https://github.com/magda-io/magda

[53] Netflix, "Genie: Distributed big data orchestration service," 2025, accessed: October 2025. [Online]. Available: https://github.com/Netflix/genie

[54] C. Delimitrou *et al.*, "DeathStarBench: An open-source benchmark suite for cloud microservices," 2025, accessed: October 2025. [Online]. Available: https://github.com/delimitrou/DeathStarBench

[55] Macrozheng, "Mall-Swarm: Microservices e-commerce system based on spring cloud," 2025, accessed: October 2025. [Online]. Available: https://github.com/macrozheng/mall-swarm

[56] MITRE, "Common vulnerabilities and exposures (cve)," 2025, accessed: October 2025. [Online]. Available: https://www.cve.org/

[57] "National vulnerability database," 2025, https://nvd.nist.gov/vuln/.

[58] newbee-ltd, "newbee-mall: A distributed e-commerce system developed with Spring Boot and Vue," 2025, accessed: October 2025. [Online]. Available: https://github.com/newbee-ltd/newbee-mall-cloud

[59] zlt2000, "ZLT Platform: A microservices platform based on spring cloud," 2025, accessed: October 2025. [Online]. Available: https://github.com/zlt2000/microservices-platform

[60] LINE Corporation, "Armeria: Your go-to microservice framework for any situation, from the creator of Netty et al," 2025, accessed: October 2025. [Online]. Available: https://github.com/line/armeria

[61] Spring Attic, "Spring Cloud Data Flow: A cloud-native orchestration service for composable microservice applications on modern runtimes," 2025, accessed: October 2025. [Online]. Available: https://github.com/spring-attic/spring-cloud-dataflow

[62] NVD, "Cve-2023-38493 detail," 2025, accessed: October 2025. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-38493

[63] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1867–1881.

[64] W. Takerngsaksiri, J. Pasuksmit, P. Thongtanunam, C. Tantithamthavorn, R. Zhang, F. Jiang, J. Li, E. Cook, K. Chen, and M. Wu, "Human-in-the-loop software development agents," in *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2025, pp. 342–352.

[65] 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 *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, May 2023.

[66] newbee-ltd, "newbee-mall: A distributed e-commerce system developed with Spring Boot and Vue," 2025, accessed: October 2025. [Online]. Available: https://github.com/newbee-ltd/newbee-mall

[67] J. Zhang, R. Piskac, E. Zhai, and T. Xu, "Static detection of silent misconfigurations with deep interaction analysis," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

[68] J. Wang, Y. Wu, G. Zhou, Y. Yu, Z. Guo, and Y. Xiong, "Scaling static taint analysis to industrial soa applications: A case study at alibaba," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1477–1486.

[69] J. Guo, C. Wang, X. Xu, Z. Su, and X. Zhang, "Repoaudit: An autonomous llm-agent for repository-level code auditing," in *Proceedings of the 42nd International Conference on Machine Learning*, 2025.

[70] Z. Li, S. Dutta, and M. Naik, "Llm-assisted static analysis for detecting security vulnerabilities," in *Proceedings of the 13th International Conference on Learning Representations (ICLR)*, Singapore, Apr. 2025.

[71] A. Lekssays, H. Mouhcine, K. Tran, T. Yu, and I. Khalil, "LLMxCPG: Context-Aware vulnerability detection through code property Graph-Guided large language models," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 489–507.

Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper presents NEO, an agentic program analysis framework for detecting privilege escalation vulnerabilities in microservice architectures. NEO combines the semantic reasoning capabilities of LLMs with the traditional static program analysis. NEO uses an LLM-driven agent to iteratively plan and perform code queries across services and languages, identifying privileged operations, tracing cross-service data flows, and checking for missing or inadequate authentication. Evaluated on 25 open-source microservice applications, NEO finds 24 vulnerabilities. NEO significantly improves detection coverage and reduces missed vulnerabilities, compared to existing tools.

A.2. Scientific Contributions

- 3. Creates a New Tool to Enable Future Science.
- 4. Addresses a Long-Known Issue.
- 5. Identifies an Impactful Vulnerability.
- 6. Provides a Valuable Step Forward in an Established Field.

A.3. Reasons for Acceptance

1) The paper presents a well-designed system that effectively integrates LLMs with static analysis, enabling scalable and targeted detection of privilege-escalation vulnerabilities in complex microservice architectures. NEO's use of structural code-search primitives avoids the limitations of large-context ingestion and allows the agent to reason over heterogeneous, cross-service codebases.
2) The evaluation is strong and comprehensive. This paper analyzed 25 real-world microservice applications, uncovering previously unknown vulnerabilities—many confirmed and patched by developers—demonstrating clear practical impact. The system also generalizes beyond its primary task, detecting other bug classes such as command injection and SQL injection with only minor prompting changes, indicating broader applicability.
3) Architecturally, NEO is well-motivated and logically structured, and its design is validated by robust experiments, diverse baselines, and well-posed research questions. The inclusion of runtime and API cost measurements adds transparency into the system's operational overhead.

A.4. Noteworthy Concerns

1) *Model and Prompt Sensitivity and Reproducibility Concerns*: NEO relies heavily on LLM reasoning and prompt design, as a result, using different LLM models may impact the results and require additional prompt tuning.
2) *Dependence on CodeQL and Its Limitations*: NEO heavily depends on CodeQL, inheriting its constraints and reducing portability to other analysis backends.