The Dark Side of Flexibility: Detecting Risky Permission Chaining Attacks in Serverless Applications

Xunqi Liu*,1, Nanzi Yang*,†,2, Chang Li1, Jinku Li†,1, Jianfeng Ma1, Kangjie Lu2
1State Key Laboratory of Integrated Services Networks, School of Cyber Engineering, Xidian University
2University of Minnesota
xunqiliu@stu.xidian.edu.cn, nzyang@stu.xidian.edu.cn, lc13700223956@gmail.com,
jkli@xidian.edu.cn, jfma@mail.xidian.edu.cn, kjlu@umn.edu

***Abstract*—Modern serverless platforms enable rapid application evolution by decoupling infrastructure from function-level development. However, this flexibility introduces a fundamental mismatch between the decentralized, function-level privilege configurations of serverless applications and the centralized cloud access control systems. We observe that this mismatch commonly incurs risky permissions of functions in serverless applications, and an attacker can chain multiple risky-permissioned functions to escalate privileges, take over the account, and even move laterally to compromise other accounts. We term such an attack a *risky permission chaining attack*.**

In this work, we propose an automated reasoning system that can detect risky permissions that are exploitable for chaining attacks. First, we root in attacker-centric modality abstraction, which explicitly captures how independent permissions from different functions and accounts can be merged into real attack chains. Based on this abstraction, we build a modality-guided detection tool that uncovers exploitable privilege chains in real-world serverless applications. We evaluate our approach across two major cloud platforms — AWS and Alibaba Cloud — by analyzing serverless applications sourced from their official, production-grade application repositories. As a result, our analysis uncovers 28 vulnerable applications, including five confirmed CVEs, six responsible vulnerability acknowledgments, and one security bounty. These findings underscore that the risky permission chaining attack is not only a theoretical risk but also a structural and exploitable threat already present in commercial serverless deployments, rooted in the fundamental mismatch between decentralized serverless applications and centralized access control models.

I. Introduction

As an increasingly adopted computing model [1], [2], [3], serverless computing allows customers to build and run applications without managing any servers [4], which is instead fully handled by cloud vendors through managed serverless platforms. As a result, customers can focus solely on composing serverless applications — modular systems built from fine-grained, event-driven functions. Famous enterprises, such as Coca-Cola, Telecom Argentina, Mitsubishi Heavy Industries, and Swire Properties, all develop and deploy their own serverless applications in production [5], [6], [7], [8]. Consequently, serverless applications have become a core component of modern cloud-based service architectures.

In a cloud serverless platform, serverless applications are deployed by different customers. Each customer manages her/his own serverless applications. This deployment model gives rise to two distinct dimensions: a vertical dimension within each account and a horizontal dimension across accounts. Vertically, each account may host multiple serverless applications, which are internally composed of event-driven functions. Besides, permission policies are attached to each function of an application to provide resource-level access control. Horizontally, cloud platforms support resource-sharing mechanisms, e.g., Lambda Layers, Amazon Elastic Container Registry (ECR) images, that allow components to be reused across accounts for efficiency and modularity. Note that in such a deployment model, permission policies play a critical role as the primary enforcement mechanism for access control across both the vertical and horizontal dimensions. Improperly scoped or inconsistently maintained policies may expose sensitive resources, violate function isolation, and ultimately undermine the security of the entire system.

The decentralized nature of serverless applications and the centralized design of existing permission management systems often lead to risky permissions at the function level. On one hand, serverless architectures employ finer-grained privilege configurations compared to traditional architectures [9], [1], [10]: each individual function is assigned its own execution role, creating a highly distributed permission landscape across multiple functions within a single application. On the other hand, traditional permission systems such as AWS Identity and Access Management (IAM) [11] or Alibaba Cloud resource access management (RAM) [12], while compatible with serverless, are primarily designed for centralized management of long-lived cloud resources (e.g., EC2 instances,

* Co-first authors.
† Co-corresponding authors.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230819
www.ndss-symposium.org

ECS instances, S3 buckets). Their configurations are largely manual, coarse-grained, and centralized, lacking native support for the fine-grained, function-level privilege assignment model of serverless applications.

Prior works on serverless access control have focused on analyzing and reducing unnecessary intra-application permissions. However, determining whether a permission is "unnecessary" or not often has limited practical meaning from the attacker's perspective. More critically, certain necessary or seemingly legitimate risky permissions can still be exploited as part of risky permission chains. These risky permissions from individual functions of different applications or accounts, though benign in isolation, can be systematically chained together by an attacker both vertically (to compromise an entire account) and horizontally (to propagate across accounts).

Specifically, we consider an attacker who compromises a single serverless function — whether through external attack vectors for internet-facing functions or internal user compromise within isolated environments [13], [14]. A compromised function may escalate its privileges either directly (e.g., by leveraging permissions that grant administrative access) or indirectly, by abusing permissions that allow it to reconfigure functions from the same or different apps to run with higher-privileged roles. Such risky permission chains allow the attacker to move vertically across applications and take control of the account.

Furthermore, serverless platforms provide dynamic resource sharing mechanisms (e.g., Lambda Layers, container images) that enable automatic code composition across account boundaries. These mechanisms create novel opportunities for how risky permissions can be abused across accounts due to dynamic resource sharing. Once an attacker compromises one account, they can exploit these shared resources to propagate malicious code or configurations to functions in other accounts. When such target functions hold risky permissions, the attacker can systematically abuse these permissions within the newly compromised environment, effectively extending their privilege escalation capabilities across account boundaries through the platform's native sharing mechanisms.

Although this insight shifts us from abstract least-privilege principles toward the detection of concretely exploitable permission chains, determining which function combinations across applications and accounts are exploitable for chaining attacks is a challenging task. This challenge stems from the complexity of serverless privilege landscapes: permissions are fragmented across hundreds of functions, roles are reused across different functions with different intents, and attack paths may span multiple accounts through dynamic resource sharing mechanisms.

Existing IAM analysis tools [15], [16], [17] typically operate at the role level, focus on identifying individual risky permissions to roles, and are scoped to single-account settings. They lack the semantic reasoning of how functions with risky permissions can be chained together to make potential exploitation paths between apps and accounts. As a result, emergent privilege escalation paths, which are composed of seemingly benign permissions, often remain invisible to traditional static analysis tools.

To address this gap, in this paper, we propose a reasoning framework from the perspective of a serverless attacker, who can potentially chain permissions across functions and accounts. We first abstract common privilege escalation behaviors into a set of formal attack modalities, and then use these modalities to guide the detection of exploitable risky permission chains in real-world serverless applications.

To systematically capture how risky permissions can be chained into practical escalation paths, we define three formal attack modalities. These modalities are derived by considering two orthogonal dimensions from the attacker aspect: (1) the attacker's capability in a single account — whether they can escalate privileges directly or indirectly (e.g., by leveraging other functions); and (2) the serverless architecture topology — whether the escalation occurs within an account (vertical) or across accounts (horizontal). Based on the two dimensions, we abstract three formal attack modalities. *First*, in the direct-vertical case, the attacker-controlled function possesses risky permissions and can directly escalate to take over the account (modality 1). *Second*, in the indirect-vertical case, the function lacks direct escalation capability, but can indirectly escalate by creating or reconfiguring other functions to run under higher-privileged roles (modality 2). *Third*, in the horizontal case, after compromising one account, the attacker leverages cross-account resource sharing and risky permissions to propagate laterally into other accounts (modality 3). While two dimensions could yield four combinations, in practice, the semantics of horizontal escalation do not meaningfully distinguish between direct and indirect patterns. Therefore, we abstract horizontal escalation as a unified modality and define three representative attack modalities in total.

To translate formal attack modalities into practical implementation, we design a two-phase detection framework based on a filtering-and-instantiation paradigm. The key insight is to treat the detection problem as a form of privilege vector space pruning. More specifically, we first reduce the search space by filtering out permissions that are unlikely to participate in compositional escalation chains. Then, we apply structured modalities to identify meaningful escalation chains. This "filter-and-instantiate" approach enables our system to scale to real-world deployments while preserving semantic fidelity in capturing compositional attack paths.

In the first phase, we extract all execution roles, function configurations, and permission assignments from serverless deployments. We then identify functions that are granted risky permissions that are known to be exploitable in privilege escalation. This phase reconstructs the fragmented privilege landscape induced by function-scoped IAM bindings, and prunes the privilege space to retain only high-risk nodes that are likely to participate in escalation chains.

In the second phase, we apply a modality-based algorithm to evaluate whether these functions can be composed into concrete privilege escalation chains. Our modality-based exploitability reasoning approach checks for direct privilege escalation (modality 1), escalation via hijacked privileged functions (modality 2), and cross-account propagation through shared resources (modality 3). The analysis produces potential privilege escalation chains, detailing how permissions from multiple functions and accounts can compose into exploitable attack paths. As a result, our approach reconstructs full attack chains rooted under serverless semantics — capturing how benign permissions, when composed across loosely coupled functions and services, lead to emergent privilege amplification.

To evaluate the effectiveness of our approach, we test it against real-world serverless applications deployed in actual commercial cloud environments. However, large-scale, deployed, and production-grade serverless applications are often difficult to access. Among major cloud providers, only AWS and Alibaba Cloud offer officially maintained serverless application repositories [18], [19], which host reusable, customer-facing applications actively used in practice. Accordingly, for each of the two cloud vendors, we install those applications provided by their repository and run our tool to detect potential privilege escalation paths. As a result, our approach successfully identifies 28 vulnerable real applications exhibiting privilege escalation chains and covers all attack modalities. We report our findings to related developers and cloud vendors. Finally, we have obtained five CVEs, six acknowledgments, and one security bounty. Our results show that the risky permission chaining attack is both prevalent and exploitable in real-world serverless environments.

Overall, this paper makes the following contributions:

Novel attack scope: We show how serverless intrinsic function-level granularity, and cross-account resource sharing mechanisms can lead to risky permissions into broader, multi-stage escalation chains, which we term *risky permission chaining attacks*. When exploited, attackers can escalate privileges both inter-apps and inter-accounts.

Modality-based reasoning: We introduce a layer modality framework, which generalizes beyond pattern matching by modeling semantically valid privilege transitions across functions, applications, and accounts, extending the search space from single permission risk to novel chaining attacks. We open-source our tool at: https://doi.org/10.5281/zenodo.16957393.

Real security impact: By applying our tool to production-grade serverless applications deployed on AWS and Alibaba Cloud, we detect and uncover multiple previously unknown privilege escalation vulnerabilities, with five new CVEs, several acknowledgments, and a security bounty.

II. Background

In this section, we present the necessary background knowledge of our work, including the serverless applications, architecture, and access control.

A. Serverless Applications

Serverless applications are the central abstraction and operational unit in serverless architecture, built and executed atop serverless platforms provided by major cloud vendors. Rather than managing underlying infrastructures intrinsic to traditional cloud services, developers only focus on implementing business logic in lightweight functions or directly deploying pre-built serverless applications from repositories. These applications are executed within managed serverless platforms provided by major cloud vendors such as AWS, Microsoft Azure, Google Cloud, IBM Cloud, and Alibaba Cloud [20], [21], [22], [23], [24], enabling rapid development and scalable deployment.

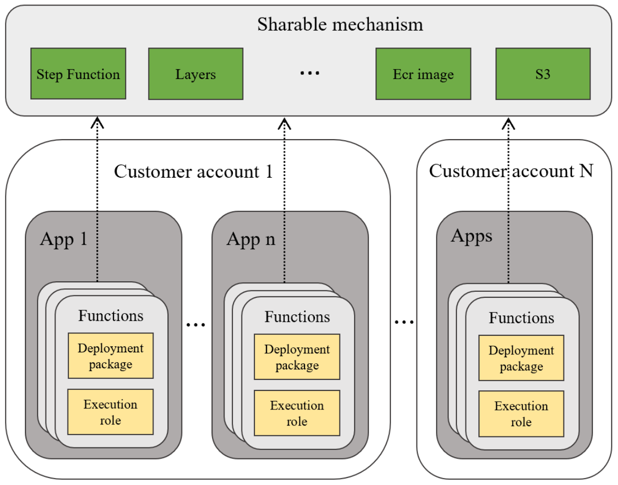

Fig. 1: Serverless application architecture.

To accelerate adoption, cloud vendors support not only customer-developed application development, but also application-level reuse through application repositories. Among major platforms, only AWS and Alibaba Cloud provide production-grade serverless application repositories — the AWS Serverless Application Repository (SAR) and Alibaba Serverless Application Center — where customers can directly deploy reusable applications maintained by cloud vendors or third-party developers [25], [19]. These repositories play a critical role in our study: they host realistic, production-facing applications that reflect how serverless functions deploy, update, and assign permissions in practice. Compared to toy examples offered by other platforms [26], [27], [28], these applications exhibit complex permission bindings, cross-function interactions, and shared resources — making them suitable targets to analyze exploitable risky permission chains in serverless applications. Therefore, we select serverless applications from these two repositories as the primary data source for our analysis and tool evaluation (see §V). The application structure and permission assignments that they reveal also guide our following architectural model and permission management.

B. Serverless Application Architecture

Serverless applications adopt a function-centric architecture. As shown in Figure 1, there are multiple serverless applications deployed by different customers. A typical serverless application (or *app*) is composed of multiple functions, each representing a discrete unit of code execution. More specifically, a typical function consists of the deployment package [29] and execution role [30]. The deployment package contains the code and dependencies needed to perform its functionalities. And the execution role is the permissions granted to this function by developers, which determine which other services or resources this function can interact with.

Besides, serverless platforms promote resource-sharing mechanisms to optimize code reuse and deployment efficiency. These mechanisms (e.g., Lambda layers [31], ECR images [32], etc.) enable multiple functions or even functions across different accounts to share the same code and dependencies on demand. Lambda Layers and container images exhibit automatic execution behavior that distinguishes them from traditional resource sharing. When a function attaches a Lambda Layer or uses a shared container image, any code within these resources is automatically loaded and executed during function initialization without explicit user intervention.

While beneficial for development efficiency, these mechanisms introduce a cross-tenant attack surface when combined with risky function-level permissions. More specifically, a shared resource from one account can automatically propagate code execution to other tenants or applications, enabling attackers to establish footholds across multiple accounts through the platform's native sharing infrastructure.

C. Access Control

Access control plays a critical role in securing serverless applications. To support access control, cloud providers interact serverless applications with existing access management systems, such as AWS Identity and Access Management (IAM) [11] and Alibaba Cloud Resource Access Management (RAM) [12]. These two mechanisms follow the same principle.

Instead of granting permissions directly to functions, developers define execution roles with attached policies, and assign them to functions at deployment. In practice, this decoupling, while flexible, leads to role reuse and permission over board across multiple functions. As functions are added or modified over time, the shared roles often accumulate privileges to accommodate diverse execution needs. This pattern significantly expands the privilege surface and creates opportunities for unexpected permission composition.

More specifically, each function in a serverless environment is bound to an execution role defined in IAM (or RAM), which governs its operations on resources via attached policies. For example, the `aws-deployment-framework`, a serverless application developed by AWS, contains six functions: StackWaiter, DetermineEventFunction, RoleStackDeploymentFunction, UpdateResourcePoliciesFunction, MovedToRootActionFunction, and CrossAccountExecuteFunction. All of them share the same execution role `LambdaRole`. As shown in Figure 2, the StackWaiter function (line 4) is assigned an execution role named LambdaRole (line 5). This role has a customer-managed [33] policy named LambdaPolicy attached to it (line 33). This policy includes the "sts:AssumeRole" action of "*" resources (line 29), which allows the StackWaiter to assume any role within the customer's account, including the *admin* role.

```
1  StackWaiterFunction:
2      Type: "AWS::Serverless::Function"
3      Properties:
4        FunctionName: StackWaiter
5        Role: !GetAtt LambdaRole.Arn
6        ...
7  ...
8  LambdaRole:
9      Type: "AWS::IAM::Role"
10     Properties:
11       AssumeRolePolicyDocument:
12         Version: "2012-10-17"
13         Statement:
14           - Effect: "Allow"
15             Principal:
16               Service:
17                 ...
18               - "lambda.amazonaws.com"
19             ...
20 ...
21 LambdaPolicy:
22     Type: "AWS::IAM::ManagedPolicy"
23     Properties:
24       PolicyDocument:
25         Version: "2012-10-17"
26         Statement:
27           - Effect: "Allow"
28             Action:
29               - "sts:AssumeRole"
30               ...
31             Resource: "*"
32       Roles:
33         - !Ref LambdaRole
34
```

Fig. 2: The `StackWaiter` function.

Customers typically manage execution roles and policies at deployment time, but rarely reason about how these permissions may be composed and abused from an attacker's perspective. While role reuse simplifies deployment, it often leads to risky permissions to individual functions that appear benign in isolation. However, in large-scale serverless deployments, these risky permissions can become exploitable when systematically chained across functions, applications, and accounts.

Specifically, when multiple serverless applications coexist in a single cloud account, disjoint risky permissions from different apps may be inadvertently composable. This enables inter-application privilege escalation paths, where permissions intended for one application are unintentionally exposed to or leveraged by others, forming what we define as exploitable chains. Even worse, when these risky permission roles are combined with cross-account resource sharing mechanisms (e.g., Lambda layers), attackers may further escalate their reach to compromise resources in other accounts — creating a multi-layered attack surface that spans multiple functions, applications, and even cloud tenants. We discuss more in §III.

III. Motivation

A. Threat Model

In this paper, we assume an attacker who gains control over a single compromised function and aims to escalate her/his privilege within the customer's cloud account. Beyond this, the attacker can even pursue cross-account privilege escalation to compromise other accounts under the same cloud provider.

We consider two distinct attack scenarios that reflect realistic serverless deployment scenarios. First, for Internet-facing functions that are directly accessible from external networks, outside attackers can exploit vulnerabilities to achieve function-level compromise [13], [34], [35], [36], [37]. Second, for functions deployed in isolated environments without direct Internet connectivity, compromise can occur through internal attack vectors such as malicious insiders. In both scenarios, once initial function compromise is achieved, our privilege escalation analysis applies regardless of the function's network accessibility.

In contrast to the weak initial capability, the attacker's potential impact is significant. We demonstrate that compromising a single function can lead to intra-account privilege escalation and even cross-account compromise, highlighting the disproportionate security risks in serverless environments.

B. Attack Definition

Prior works on serverless access control have focused on analyzing and reducing unnecessary intra-application permissions. However, determining whether a permission is "unnecessary" or not often has limited practical meaning from the attacker's perspective. More critically, certain necessary or seemingly legitimate permissions can still be exploited when systematically composed across functions from different apps and accounts. This insight shifts us from abstract least-privilege principles toward the detection of concretely exploitable permission chains.

Risky permission chaining attack: A risky permission chaining attack occurs when an attacker compromises a function and systematically composes risky permissions from different functions, applications, and accounts into exploitable escalation paths that span privilege boundaries.

Existing concepts such as "IAM misconfiguration" or "multi-IAM attacks" [38] fall short of describing this risk, as they focus on static policy mistakes or overly permissive roles. In contrast, risky permission chaining attacks arise from a structural insight: the ability of attackers to weaponize the risky permissions in serverless systems to transform isolated permissions into exploitable paths.

C. Motivating Example: A Real Attack in AWS

We now illustrate a real-world multi-stage, cross-account risky permission chaining attack observed in AWS. This attack emerges from the composition of loosely-scoped permissions and the sharing semantics of serverless platforms, forming executable attack chains across functions and cloud tenants. This case, which affected production applications from AWS and Coralogix, has been acknowledged via a CVE and a public patch.

Detailed analysis: We consider a real scenario where two serverless accounts have different applications. As shown in Figure 3, the customer AWS account A has installed the *aws-deployment-framework* (*ADF* for short) application. The AWS account B has installed the Coralogix-Lambda-Manager and ADF applications. As we discussed in §II-C, the attacker who controls one of ADF's functions (e.g., the StackWaiter) can abuse the "sts:AssumeRole" of "*" to assume the *admin* role within the customer's account. As a result, the attacker can escalate vertically and take over the customer account A (① in Figure 3).

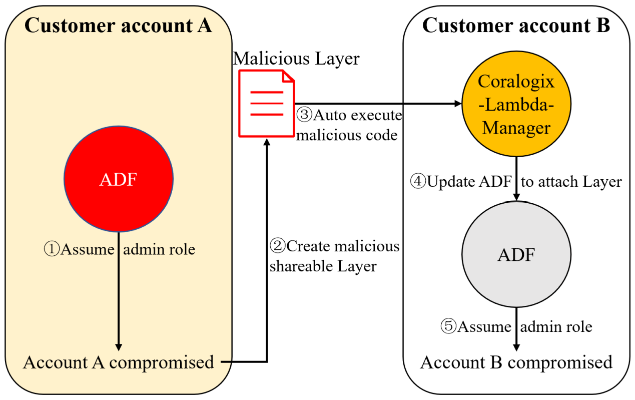

Fig. 3: A real attack in AWS.

Based on this, the attacker further leverages the compromised account A to create a malicious Lambda Layer [31] embedded with backdoor code (② in Figure 3). The AWS Lambda Layer mechanism exhibits two features that make it an effective attack vector. First, it can be used to share and attach an executable code module across serverless functions in different accounts. Second, once a layer is attached to a function, any module imported from the layer will be executed automatically as part of the function's initialization process. As a result, if a function in account B (e.g., Coralogix-Lambda-Manager) attaches this malicious layer — intentionally or inadvertently — the embedded backdoor logic will be triggered automatically during function startup. This enables the attacker to escalate horizontally and gain a function-level foothold in account B (③ in Figure 3).

Moreover, the Coralogix-Lambda-Manager's function in account B also has risky permissions that can be exploited to compromise this account. More specifically, it is granted "lambda:UpdateFunctionConfiguration" of "${AWS::AccountId}:function:*", which allows the attacker to reconfigure any function in the account — including ADF — to attach the same malicious layer and control ADF's function in account B (④ in Figure 3). Similarly, as in the first stage of the attack, this enables the attacker to use the infected ADF's function to assume an *admin* role and take full control of account B (⑤ in Figure 3).

Acknowledgment: We responsibly disclosed our findings to the affected vendors. The ADF is developed by AWS, which has acknowledged the issue and assigned us a high-severity CVE (CVE-2024-37293). In addition, the Coralogix team confirmed the vulnerability in their Coralogix-Lambda-Manager application and released a public patch via a GitHub pull request. These acknowledgments demonstrate the practical impact and severity of the discovered issue.

Lessons learned: This two-stage attack illustrates the core insight behind risky permission chaining attacks: in serverless environments, removing unnecessary permissions to follow the least privilege rules can not fundamentally solve problems. From the attacker's perspective, an attacker can chain risky permissions inter-applications, no matter necessary or unnecessary, into systematic compromise — first vertically, by involving sts:AssumeRole permission and escalating within an account; then horizontally, by propagating across tenants through lambda:UpdateFunctionConfiguration and layer attachment.

Such escalation paths are neither explicitly granted nor statically visible — they emerge from the mismatch between the decentralized function with resource sharing of modern serverless deployments and the centralized feature of capable but misaligned permission management mechanisms (e.g., IAM). The resulting compromise of two real applications demonstrates that risky permission chaining attacks are not only hypothetical — they are structural, exploitable, and already deployed in the wild.

IV. Systematic Exploitability Reasoning for risky permissions

A. Detection Framework Overview

Our goal is to detect concrete exploitable instances of risky permission chains in real-world serverless deployments. However, this task is challenging due to the combinatorial nature of privilege composition: permissions are scattered across hundreds of functions and roles, often spanning multiple services and accounts. As a result, the potential space of privilege escalation paths is vast, and most paths are benign or irrelevant.

To address this issue, we design a two-phase detection framework based on a filtering-and-instantiation paradigm. The key insight is to treat the detection problem as a form of *privilege vector space pruning*. That is, we first reduce the search space by filtering out permissions that are unlikely to participate in chaining attacks. This step collapses the vast search space into a tractable set of high-risk privilege nodes. Then we apply a structured modality reasoning algorithm to identify meaningful escalation chains. This "filter-and-instantiate" approach enables our system to scale to real-world deployments while preserving semantic fidelity in capturing compositional attack paths.

In the first phase, we extract all execution roles and policy bindings, and identify functions granted sensitive permissions that are known to be exploitable in privilege escalation. This produces a refined privilege graph that retains only attacker-relevant operations and potential entry points (i.e., *risky permission identification* in Figure 4).

In the second phase, we instantiate our formal attack modalities as permission composition algorithms. It encode the structural and semantic requirements of risky permission chaining attacks — such as role assumptions, function reconfigurations, or cross-account infection — and search for valid paths in the refined graph that satisfy these patterns. This enables us to detect concrete, multi-stage exploitable paths from the attacker aspect (i.e., *modality-based exploitability reasoning* in Figure 4).

B. Risky Permission Identification

TABLE I: Capability Comparison of IAM Analysis Tools

Tool name	**Risk detection**	**Attack chains**	**Cross account**
IAMGraph	✗	✗	✗
IAMSpy	✗		
PMapper	✓		
Cloudsplaining	✓		
Red-Shadow	✗		
AWS Access Analyzer	✓		
AWS Policy Simulator	✗		
Our Approach	✓	✓	✓

While our analysis focuses on reasoning the exploitable risky permission chains in serverless scenarios, we intentionally build on existing research to define sensitive IAM actions. These well-established sets of high-risk permissions provide a reliable foundation for filtering out unlikely escalation paths, allowing us to concentrate on the unique privilege propagation behaviors enabled by serverless systems, such as function reconfiguration, implicit role chaining, and cross-account propagation. Our contribution, therefore, does not lie in redefining what permissions are dangerous, but in showing how risky permissions can be composed into novel escalation paths that are specific to serverless architectures and typically invisible to traditional IAM analyses.

Concretely, we scan all execution roles in the target cloud environments and extract their associated policies. We then flag as "risky" for any role that includes one or more actions from the pre-defined sensitive permission set [38], [39], [40]. These permissions are summarized in Table V and Table VI. As a result, we match each risky role to the serverless functions that actually assume it at runtime. This role-to-function mapping enables us to localize potential escalation points to concrete execution targets, providing the necessary entry points for subsequent modality-based identification.

C. Exploitability Analysis for Risky Permissions

After identifying functions with risky permissions, we proceed to determine whether they can be composed into exploitable paths. To achieve this, we adopt a layer-based detection approach guided by our formal attack modalities. Each modality is instantiated as a structured detection pattern that captures the semantic and structural preconditions of a specific class of privilege escalation.

1) Challenges: To systematically analyze whether risky permissions are susceptible to chaining attacks in serverless applications, we must move beyond ad-hoc case studies and instead develop a generalizable analysis approach. While existing IAM analysis tools [17], [41], [42], [15], [43], [16], [44] have proven effective for identifying individual risky permissions within single accounts, they exhibit fundamental limitations when applied to serverless environments.

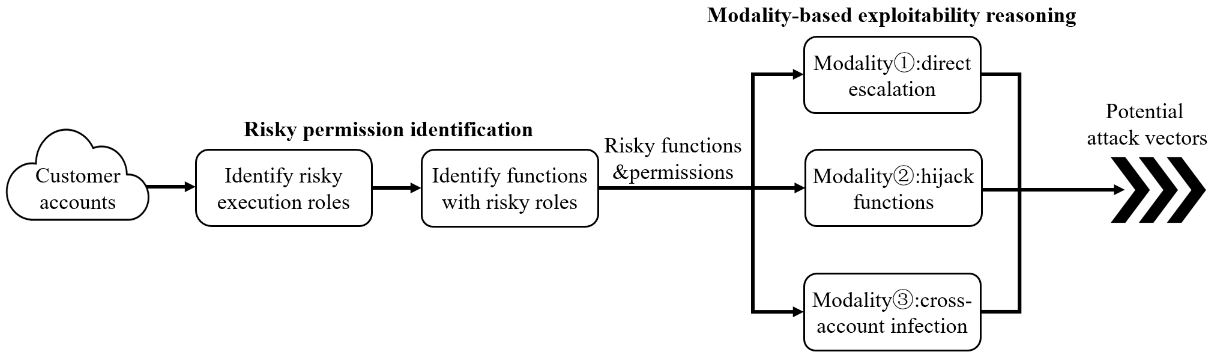
Modality-based exploitability reasoning
Risky permission identification
Customer accounts
Identify risky execution roles
Identify functions with risky roles
Risky functions &permissions
Modality①:direct escalation
Modality②:hijack functions
Modality③:cross-account infection
Potential attack vectors

Fig. 4: Detection framework overview.

As summarized in Table I, we conducted a systematic comparison of seven existing tools across three key capabilities: risky permission detection (*Risk detection*), attack chain detection (*Attack chains*), and cross-account detection support (*Cross account*). Our experiments reveal that while some tools (e.g., Cloudsplaining, PMapper, AWS Access Analyzer) can successfully detect individual privilege escalation risks, none support multi-stage attack chain analysis or cross-account privilege propagation — two critical attack vectors in modern serverless deployments. This limitation stems from their design focus on role-level policy analysis, without accounting for how permissions are actually composed, reused, and propagated across functions, applications, and accounts in serverless environments.

To address these limitations, we need to extract and generalize formal attack modalities by systematically analyzing real-world cases from the attacker's perspective. After that, we implement these attack modalities as a structured detection algorithm to reason the potential exploitation paths in real-world accounts. Each step is non-trivial and raises distinct technical challenges:

Challenge 1: It is hard to build formal attack modalities beyond individual attacks.

Systematically detecting risky permission chaining attacks requires generalizing beyond individual case studies. Prior works — both academic and industry [38], [40], [39], [45] — have uncovered concrete examples of privilege escalation in serverless applications. However, these efforts remain case-by-case, lacking a unified model for reasoning about how such attacks arise. Without such abstraction, the detection is limited to manually discovered instances and cannot scale to unseen deployments.

In serverless systems, privileges are assigned at the granularity of individual functions — each bound to its own execution role and policies. Yet risky permission chaining attacks emerge from privilege chains that span multiple roles, functions, and even accounts. This creates a fundamental mismatch in analysis granularity: the privilege boundary is local, while the attack surface is global. Capturing these behaviors in a reusable model requires reasoning about how isolated permissions can be composed into emergent privilege escalation paths — across loosely coupled components that evolve independently.

To solve this challenge, our key observation is that risky permission escalation behaviors exhibit recurring structural patterns, which can be systematically modeled. We elaborate on this model in §IV-C2.

Challenge 2: It is hard to realize attack modalities in tools to identify real exploitable paths.

While attack modalities offer a structured abstraction for modeling our attack, they cannot identify actual attack instances on their own. Real-world detection requires instantiating these modalities over the concrete configurations of cloud deployments, i.e., roles, policies, functions, and their relationships. Unfortunately, existing IAM tools are not designed for this: they operate at the level of individual roles or policies, and cannot reason about cross-function, cross-account privilege composition. To bridge this gap, a dedicated analysis mechanism is needed.

To solve this challenge, our key observation is that attack modalities exhibit a natural hierarchical structure that enables layered reasoning about privilege escalation paths. Each modality can be formalized as a set of structural and semantic patterns, where higher-level modalities build upon lower-level ones to form increasingly sophisticated attack chains. This enables us to transform the exploitability reasoning problem into a layer-based identification task — searching for attacker-reachable privilege chains that match each mode under serverless semantics. We describe more details in §IV-C3.

2) Attack modalities modeling: As we discussed in the first challenge of §IV-C1, systematically reasoning the exploitation paths of risky permissions first requires abstracting the core attack behaviors into generalizable modalities. To achieve this, our key observation is that the privilege escalation risks can be abstracted along two orthogonal dimensions. The first is *attacker capability* — whether a function under attacker control can directly perform privilege escalation (direct), or needs to rely on compromising additional functions (indirect). The second is the *topology of privilege propagation* — whether escalation occurs within a single account (vertical) or spans across multiple accounts (horizontal).

Combining these dimensions, we observe three formal attack modalities that are repeatedly exploitable in real-world deployments. First, an attacker can escape from a function and

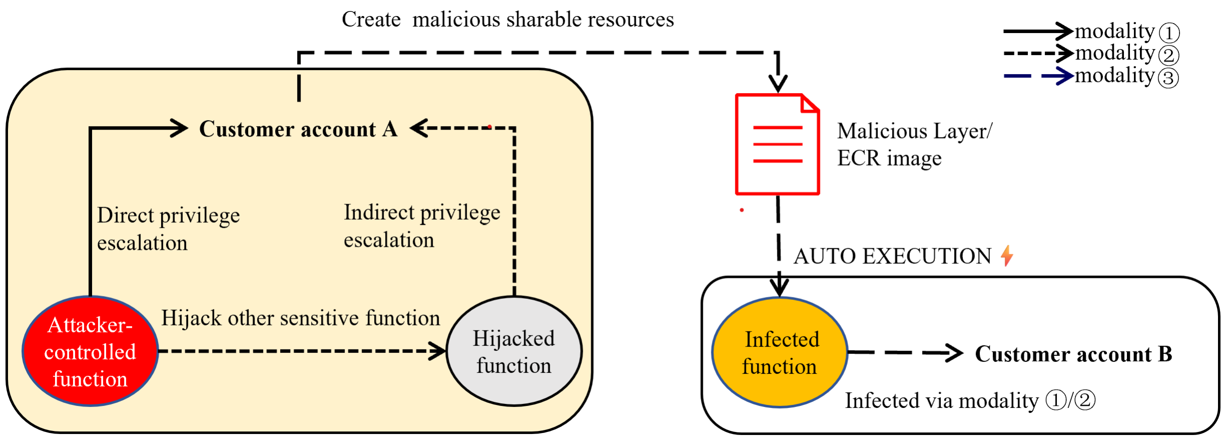
Create malicious sharable resources
modality ①
modality ②
modality ③
Customer account A
Direct privilege escalation
Indirect privilege escalation
Hijack other sensitive function
Attacker-controlled function
Hijacked function
Malicious Layer/ ECR image
AUTO EXECUTION
Infected function
Customer account B
Infected via modality ①/②

Fig. 5: Attack modalities of the risky permission chaining attack.

compromise a single account by exploiting critical permissions directly (① in Figure 5). Second, the attacker can abuse the permissions of a compromised function to hijack other critical functions, and then abuse the new granted critical permissions to take over the account indirectly (② in Figure 5). Third, serverless platforms enable horizontal privilege propagation via cross-account sharing of components such as Lambda Layers; once an attacker leverages a shared resource and compromises a function in a new account, previously used vertical escalation techniques can be recursively applied, allowing the attacker to expand control across multiple accounts (③ in Figure 5).

Notably, the horizontal propagation modality presumes that the attacker has already compromised an account via vertical escalation. It focuses on cross-tenant movement — how attackers reuse their foothold to infect other accounts — rather than on the method of escalation itself. As such, it does not introduce a separate attacker capability dimension, but rather extends the impact of existing vertical modalities across accounts. This design results in three representative modalities, as shown in Figure 5, instead of a full 2×2 combination.

a) Obtaining account admin directly: Our first modality is to abuse the risky permission of a serverless function to directly take over the entire account. Specifically, if the attacker-controlled function is granted permissions that allow it to escalate to administrative privileges (e.g., assuming the admin role, attaching the AdministratorAccess policy, etc.), the attacker can abuse the function to escalate privileges without additional steps. This pattern is demonstrated in the first stage of our motivating example. The attacker-controlled ADF's function leverages the "sts:AssumeRole" permission of "*" to directly obtain the admin role and take over the customer account A.

b) Hijacking critical function in the same account: Our second modality is to abuse the permissions of a compromised function to hijack other critical functions, and then exploit their critical permissions to take over the account indirectly.

For example, api-gateway-dev-portal (AGDP for short) is another AWS serverless application, and one of its functions has an execution role that is granted both "iam:PassRole" and "lambda:CreateFunction". If an attacker gains control of this function, he/she can use it to create a new Lambda function and specify the execution role of another existing high-privilege function (e.g., from the ADF application) during creation. Once the new function is deployed, it inherits the "sts:AssumeRole" capability from the ADF's function's execution role, enabling the attacker to escalate privileges without accessing the ADF's function directly.

c) Infecting other accounts via resource sharing mechanism: Our third modality extends the attack surface across account boundaries by abusing serverless platforms' support for cross-account resource sharing. An attacker who compromises one account based on the former two modalities can craft and share a malicious payload with other accounts to infect their functions. Once a foothold is established in the new account, previously described privilege escalation modalities can be recursively applied to control other accounts horizontally. This pattern is demonstrated in the second stage of our motivating example, where a malicious layer created from account A is used to compromise account B, eventually allowing the attacker to take over both accounts.

3) Modality-based exploitability analysis: To detail our analysis approach, we first discuss the attack graph construction. Based on that, we give a detailed algorithm analysis.

Attack graph construction: Our system is designed with scalability and completeness in mind via a two-stage node-edge analysis.

In our attack graph, each node represents a function that holds at least one risky permission. To identify such nodes, we traverse all IAM roles and policies, and associate each function with its execution role. As a result, we can extract permissions granted to each function and build the node. To avoid a combinatorial explosion, we prune the graph by retaining only functions with risky permissions, and only include those risky permissions per function.

In the attack graph, each edge represents a potential privilege escalation chain from one function to another, constructed based on our layered modality framework. Each modality defines a specific class of escalation logic used to determine whether an edge should be added between two functions.

Algorithm 1: Modality-Based Exploitable Path Detection

```
Input: A set of accounts $A = \{A_1, A_2, ..., A_n\}$, each
       with functions and execution roles
Output: Three sets of attack paths:
        Modality ①Set, Modality ②Set,
        Modality ③Set
1  foreach account acc in A do
2      foreach function f in acc.functions do
3          if $f.permissions \supseteq$ RequiredPermissions ① then
4              Modality ①Set[acc] ← Modality ①Set[acc] ∪ $\{f\}$
5          end
6      end
7      foreach function f in acc.functions do
8          if $f.permissions \supseteq$ HijackPermissions then
9              foreach $f_{target}$ in Modality ①Set[acc] do
10                 if CanHijack($f$, $f_{target}$) then
11                     Modality ②Set[acc] ← Modality ②Set[acc] ∪ $\{(f, f_{target})\}$
12                 end
13             end
14         end
15     end
16 end
17 foreach account pair $(accA, accB)$ in $A \times A$, $accA \neq accB$ do
18     if Modality ①Set[accA] $\neq \emptyset$ or Modality ②Set[accA] $\neq \emptyset$ then
19         if HasFunctionWithInfectionCap($accB$) and (Modality ①Set[accB] $\neq \emptyset$ or Modality ②Set[accB] $\neq \emptyset$) then
20             Modality ③Set ← Modality ③Set ∪ $\{accA \rightarrow accB\}$
21         end
22     end
23 end
```

More specifically, modality ① adds an edge when a function's risky permissions can be directly abused to perform escalation. Modality ② adds an edge when a function controlled by the attacker can hijack another function that satisfies Modality ①, combining hijack and privilege actions into a compound escalation chain. Modality ③ adds a cross-account edge when a function in the source account satisfies modality ① or ② and holds cross-account access permissions. The target account must also contain functions satisfying modality ① or ②, allowing the attack chain to continue after the transition. This layered logic ensures that the algorithm remains tractable across multiple functions/accounts, while the modality semantics ensure comprehensive coverage of potential escalation chains.

Algorithm analysis: To achieve this graph-based construction, we design a hierarchical inference engine to reason compositionally and recursively, uncovering multi-stage exploit paths that would be missed by flat or isolated analysis approaches.

More specifically, modality ① serves as the foundational layer in our hierarchical inference framework. It corresponds to a single-function privilege escalation condition: if a function possesses critical permissions such as `sts:AssumeRole`, the engine directly marks it as an escalation point (lines 2-6 in Algorithm 1). Building on this, modality ② represents the next level in attacker capability: it captures a two-stage escalation path in which an attacker first hijacks a function that satisfies modality ①, and then leverages its permissions to escalate privileges further. Accordingly, our engine first identifies functions that match modality ① (line 9 in Algorithm 1), and then searches for candidate functions with hijacking permissions (lines 10-12 in Algorithm 1).

At the highest level of our hierarchical inference framework, modality ③ captures cross-account privilege propagation. This attack path begins with local privilege escalation in a source account (account A), achieved via modality ① or ②. The attacker then attempts to compromise a second account (account B) by leveraging shared resources, such as Lambda Layers, that enable code execution across account boundaries.

To be vulnerable, account B must contain at least one function that, once compromised, has the ability to modify or overwrite other functions within the same account, e.g., via the permission "lambda:UpdateFunctionConfiguration." This infection capability allows the attacker to establish a foothold in account B and recursively trigger modality ① or ② within the new account, effectively extending the privilege escalation chain across accounts.

To detect such paths, our system first checks whether both the source and target accounts contain valid local escalation chains (line 17 in Algorithm 1). Then it invokes the predicate function `HasFunctionWithInfectionCapability(accB)` to determine whether account B includes any function that, once infected, can propagate the infection to other functions via permissions such as lambda:UpdateFunctionConfiguration (line 19 in Algorithm 1). If all conditions are satisfied, the engine concludes that cross-account escalation is achievable (line 20 in Algorithm 1), and continues the recursive inference process within account B.

This recursive, multi-account reasoning completes the top layer of our hierarchical model, enabling the detection engine to uncover transitive, cross-tenant privilege escalation chains that go beyond traditional per-role or per-account analysis. For example, consider the real-world attack scenario we discussed in §III-C. In this case, account A has deployed the ADF application, and account B hosts both the Coralogix-Lambda-Manager and ADF applications. Our tool identifies a risky execution role `LambdaRole` in both accounts, which includes `sts:AssumeRole` permission, and maps it to multiple real functions (e.g., `StackWaiter`) in the ADF application. In account B, the tool specifically identifies another function in Coralogix-Lambda-Manager that holds "lambda:UpdateFunctionConfiguration" permission.

Based on these findings, the detection engine confirms that both accounts independently satisfy the conditions for modality ①, namely, privilege escalation through sensitive role permissions. Furthermore, since the function in account B allows unrestricted updates to function configurations, once it is infected by the malicious Lambda Layer deployed from account A, it can infect other functions (ADF's function) and assume the admin role. This cross-account propagation path satisfies the criteria for modality ③. This confirms the practical applicability of our modality-guided detection approach in identifying multi-stage, cross-account exploitation paths.

V. Evaluation

To evaluate the effectiveness of our detection system and assess the real-world security impact of risky permission chaining attacks in production-grade applications, we deploy and analyze applications from official repositories provided by cloud vendors. Through responsible disclosure, we have obtained 5 CVEs, one security bounty, and six confirmations from two cloud vendors, demonstrating both the practicality and security relevance of the identified risks.

A. Evaluation Scope and Methodology

For the evaluation scope, since our detection system is the first to target risky permission abuse in serverless applications, there is no existing tool or benchmark dataset to serve as a direct baseline for comparison. Moreover, due to the complexity of privilege composition across functions and accounts, it is difficult to construct synthetic ground truth datasets that comprehensively cover all possible attack paths.

Therefore, we evaluate our system by applying it to a wide range of real-world serverless applications deployed on public cloud platforms. Our evaluation focuses on the system's capability to uncover concrete attack chains in the wild, demonstrating both its practical effectiveness and the severity of our attack.

For the evaluation methodology, since our goal is to assess the existence and severity of risky permission chaining attacks in real-world serverless applications, we should avoid using toy examples that do not reflect realistic serverless configurations. Furthermore, we should evaluate those apps officially recommended or maintained by cloud vendors, as these applications are more likely to reflect production-grade configurations and widely used deployment practices. In contrast, applications from public aggregators or open-source communities (e.g., Serverless Framework [46], Awesome Serverless [47], OpenFass FunctionStore [48], etc.) often vary in quality and may not be representative of real-world deployments.

Therefore, we first select and install applications from the official serverless applications repositories provided by major cloud vendors (i.e., AWS Serverless Application Repository [18] and Alibaba Cloud Serverless Application Center [19]). Our detection system is then applied to analyze each application and extract potential issues and associated attack chains. Finally, we privately report each confirmed security issue to the corresponding vendors through responsible disclosure channels. This methodology enables us to evaluate our system's effectiveness in identifying meaningful, exploitable security risks in real-world serverless applications while maintaining ethical and responsible research practices.

TABLE II: Result summary of serverless applications.

Cloud vendor	Serverless apps	Identified apps	Confirmed apps
AWS	308	26	10
Alibaba Cloud	55	2	2

B. Real-World Attack Discovery

Rather than describing individual cases in detail, we organize our findings to demonstrate systematic patterns. Since modalities ① and ③ were illustrated in §III, we only describe the technical details and result summary of different cloud environments. In addition, we discuss one modality ② case to complete the coverage.

Ethical Considerations: All experiments in this section were conducted under strict ethical guidelines to ensure no harm to real-world systems or users. We operated entirely within cloud accounts that are fully owned and controlled by us. All identified risks were responsibly disclosed to the respective vendors before any public discussion. For more detailed discussion of ethical procedures, see VIII.

AWS result summary: There are thousands of apps provided by AWS. To ensure that our analysis targets both popular and security-critical applications, we focus on those that are both widely deployed and explicitly configure their custom access control policies. More specifically, applications with high deployment counts are more likely to be used in production settings, and custom IAM policies introduce a greater risk of risky permission chaining attacks compared to default platform configurations.

Based on these insights, we selected serverless applications from the AWS Serverless Application Repository based on two criteria: (1) the application has been deployed more than 10 times, and (2) it defines custom IAM roles or resource policies. Applying these criteria, we analyzed 308 applications and identified 26 ones with potentially exploitable issues (listed in Table II). Among these, 10 applications (7 developed by AWS and 3 by third-party developers) were confirmed by their developers through responsible disclosure, including four assigned CVEs. See detailed results summary in Table III.

Alibaba Cloud result summary: Compared to AWS, Alibaba Cloud's Serverless Application Center hosts a smaller number of serverless applications and does not expose public deployment metrics or detailed IAM configurations in application metadata. As a result, we adopted an exhaustive analysis approach and installed all 55 available applications listed in the repository.

TABLE III: Result summary of risky permission chaining attacks via serverless apps.

Developer	Serverless app	Function	Execution role	Attached policy	Permission	Modality	Disclosure status
AWS	aws-deployment-framework	StackWaiter	LambdaRole	LambdaPolicy	sts:AssumeRole of *	①	CVE-2024-37293
	measure-cold-start	Loop	LoopRole	LoopRolePolicy0	lambda:UpdateFunction Configuration of *	③	CVE-2025-45471
	autodeploy-layer	DeployToExisting Functions	DeployToExisting FunctionsRole	DeployToExisting FunctionsRole Policy0	lambda:UpdateFunction Configuration of *	③	CVE-2025-45472
	experimental-programmatic-access-ccft	ExtractCarbon EmissionsFunction	ExtractCarbon Emissions FunctionRole	ExtractCarbon EmissionsFunction RolePolicy0	sts:AssumeRole of *	①	confirmed
	aws-lambda-ecs-run-task	rLambdaFunction	rLambda FunctionRole	Administrator Access	* of *	①	
	cloudFront Extensions Console	cf_config_ version_manager	CloudFront ConfigVersion	ConfigManager	iam:CreateRole and iam:AttachRolePolicy of *	①	
	api-gateway-dev-portal	CloudFront Security HeadersLambda	CloudFrontEdge ReplicatorRole	root	iam:PassRole and lambda:CreateFunction of *	②	
LoadZilla	LoadLogic	LogicLoadEc2 DeployLambda	LogicLoadEc2 DeployLambda Role	LogicLoadEc2 DeployLambda RolePolicy0	sts:AssumeRole of *	①	CVE-2024-46511
Salesforce Service Cloud Voice	ServiceCloud VoiceLambdas	ContactLens KinesisStream AssociateFunction	KinesisStream StorageAssociate FunctionRole	SCVConnect Configurator RolePolicy	iam:PutRolePolicy of *	①	confirmed
Coralogix	Coralogix-Lambda-Manager	LambdaFunction	LambdaFunction Role	LambdaFunction RolePolicy0	lambda:UpdateFunction Configuration of ${AWS::AccountId}: function:*	③	
Alibaba Cloud	fc-llm-api	llm-server	fcdeployrole	AliyunFCFull Access	ram:PassRole and fc:* of *	②	security bounty
	fc-stable-diffusion-plus	sd	AliyunFCServer lessDevsRole	AliyunFCFull Access	ram:PassRole and fc:* of *	②	CVE-2025-45468

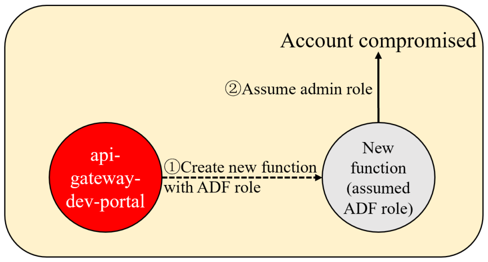
Account compromised
②Assume admin role
api-gateway-dev-portal
①Create new function with ADF role
New function (assumed ADF role)

Fig. 6: AGDP attack in AWS.

Among these, our tool identified 2 applications with potentially exploitable paths (listed in Table II). Notably, one of them was developed by Alibaba Cloud itself and has been acknowledged via Alibaba Security Response Center [49] with a bounty awarded. And the other one assigned us a new CVE (i.e., CVE-2025-45468). The risky permissions are listed in Table III.

Modality ② case: api-gateway-dev-portal: This application is a serverless app developed by AWS. It brings a function called CloudFrontSecurityHeadersLambda and carries "iam:PassRole" and "lambda:CreateFunction" of "*" resource in version `4.0.0`. An attacker can create a new Lambda function with arbitrary code and assign it a high-privilege role using `iam:PassRole`. After that, by invoking the newly created function, the attacker can indirectly escalate privileges within the account (modality ②).

More specifically, as shown in Figure 6, the attacker can leverage this privilege combination to launch a two-step escalation. First, the attacker uses the compromised AGDP function to create a new Lambda function, and during creation, specifies the high-privilege execution role of an existing function, such as one from the aws-deployment-framework (ADF), which possesses the permission `sts:AssumeRole:*` (① in Figure 6). Then, the attacker invokes the newly created function, which runs under the stolen role and gains access to its permissions. Through this role, the function can assume the admin role and take over the account (② in Figure 6). Notably, this attack achieves indirect privilege escalation without ever directly interacting with the original high-privilege ADF's function. We responsibly reported this issue to AWS, and they confirmed the underlying permission misuse.

Modality ablations: Our evaluation demonstrates the modality-based framework successfully identified recurring vulnerability patterns across multiple applications, covering all three modalities proposed by us. Furthermore, our ablation experiments show that removing any modality leads to a measurable drop in the number of detected vulnerable applications. More specifically, removing modality ③ reduce the identification of 7 cases, removing modality ② reduce the identification of 4 cases, and removing ① reduces the identification of all cases, which serve as the basic unit of our layered modality approach. This result confirms that each modality contributes uniquely to uncovering multi-stage privilege escalation chains.

C. Real-World Tool Deployment

These CVEs, bounty, and multiple confirms demonstrate how we, as researchers, apply our detection framework to analyze security risks. However, for end users, several practical aspects regarding the deployment and use of the tool in real-world settings remain unclear. More specifically, key questions may arise from end users regarding responsibility, usage frequency, and computational overhead of using our tool in real-world scenarios. Here, we address the first two aspects, while overhead evaluations are discussed in detail in §V-D.

TABLE IV: Scalability performance results in AWS.

Scenario	App count	Time (s)
Vulnerable only	1	22.55
	5	24.50
	10	25.06
	15	28.34
	20	31.47
	26	34.71
Mixed environment	50 (26+24)	39.92
	100 (26+74)	93.26
	200 (26+174)	146.46

First, for practical usage, according to the shared responsibility model widely accepted by cloud vendors [50], [51], [52], [53], securing application logic and configurations falls under customers (i.e., “security on the cloud”), not cloud vendors. Therefore, the primary responsibility for running our tool lies with serverless application developers and cloud tenants. Our tool serves two main usage scenarios: (1) Serverless app developers can integrate it into development workflows to eliminate privilege misconfigurations, and (2) Cloud tenants can use it to perform security audits on all installed applications, particularly those with custom IAM roles and cross-account resource-sharing capabilities.

Second, for usage frequency, based on our two primary usage scenarios, we recommend different execution patterns: (1) for serverless application developers, the tool should be executed before each application release or update to identify and eliminate risky permission chains in the development pipeline; (2) for cloud tenants, considering that many serverless applications are developed and maintained by different teams and may contain vulnerabilities, the tool should be run after installing new serverless applications to ensure no exploitable privilege escalation paths are introduced.

D. Performance Evaluation

To evaluate the scalability of our approach, we conduct experiments in both cloud environments. Specifically, for the AWS environment, which contains a larger application corpus and better reflects large-scale serverless deployments, we design two complementary experiments: (1) vulnerable application scalability, where we gradually increase the number of risky applications to assess core algorithm performance, and (2) mixed environment scalability, where we simulate realistic deployment scenarios where both vulnerable and benign applications coexist. For Alibaba Cloud, given the total repository size of 55 applications, we directly measure the analysis time for the complete mixed scenario to validate cross-platform consistency.

For the AWS environment, the performance results are summarized in Table IV. For the vulnerable applications only scenario, our algorithm demonstrates nearly linear scalability, with analysis time increasing from 22.55 seconds for a single application to 34.71 seconds for all 26 vulnerable applications. This modest time increase indicates efficient processing of security-critical applications. For the mixed AWS environment, our algorithm maintains reasonable performance with analysis time scaling from 39.92 seconds for 50 apps (26 vulnerable + 24 benign) to 146.46 seconds for 200 apps (26 vulnerable + 174 benign). For Alibaba Cloud, analyzing all 55 applications takes less than 80 seconds. Results demonstrate that our algorithm exhibits favorable scalability characteristics across different deployment scales and cloud platforms.

VI. Discussions

A. Scope Discussion

Our focus on serverless environments is motivated by two distinctive characteristics that amplify cross-account privilege escalation risks: (1) decentralized privilege configuration at the function level, and (2) automatic consumption of cross-account shared resources.

For example, in AWS, there are 7 major types of cross-account resource sharing mechanisms: Lambda Layers [31], ECR container images [54], S3 buckets [55], EventBridge event buses [56], Step Functions [57], Secrets Manager [58], and CloudFormation StackSets [59]. Among these, only Lambda Layers and ECR images enable automatic code execution in target accounts without explicit user intervention. As the real example discussed in §III, once a malicious layer is installed in the victim account, the malicious code will be executed and the account will be automatically compromised. In contrast, although ECR images can also be used in traditional EC2 environments, the images must be manually pulled and deployed by users. Furthermore, the fine-grained, function-level permission assignment dramatically increases the attack surface compared with traditional instance-level permissions in environments like EC2.

While our analysis focuses on serverless environments, we acknowledge that if we relax the requirement for automatic execution and consider manual triggering of shared resources, similar attack surfaces could exist in other environments. Consequently, our detection framework could potentially be extended to analyze privilege escalation risks in other cloud paradigms with comparable resource-sharing mechanisms.

B. Mitigation Discussion

The risky permission chaining attack poses a systemic threat in serverless environments due to its ability to compose and propagate privileges across apps and accounts. As a result, mitigation is critical to prevent exploitation in practice. Based on our findings and communications with multiple developers and cloud vendors, we propose several suggestions.

Restrict high-privilege capabilities: Developers should avoid assigning high-privilege permissions such as "AdministratorAccess", "sts:AssumeRole" on wildcard resources, or risky combinations like "iam:PassRole" and "lambda:CreateFunction" to the same function. These permissions, individually or in combination, can lead to direct or indirect privilege escalation. As observed in modalities ① and ②, such overly permissive configurations are the root cause of many real-world serverless vulnerabilities.

Apply fine-grained resource constraints: As shown in real attacks, multiple risks we discussed before are carried by granting permissions to wildcard resources (e.g., "*"). Developers should always specify "Resource" and "Condition" fields in IAM policies to prevent wildcard-based privilege abuse.

Avoid sharing high-privilege execution roles: As we illustrated in real cases, developers usually grant multiple functions with the same permission sets for rapid deployment and carry potential risks. We believe they should prevent multiple functions from sharing the same high-privilege execution role to reduce lateral movement risk.

Integrate our detection tool into CI/CD pipeline: Existing permission analysis tools focus on general IAM risks. They are not designed with serverless evolutions in mind, and often fail to capture dynamic privilege composition and cross-account escalation paths. Developers can leverage our automated detection system to identify privilege escalation paths before deployment, especially in CI/CD pipelines.

Cloud-vendor collaboration: Although the shared responsibility model claims that securing serverless applications belongs to customers, cloud vendors should provide native support to help developers reason about privilege composition. This includes surfacing privilege inheritance paths during deployment, warning against high-privilege role reuse, and enabling visualization tools for permission propagation. These platform-level capabilities, integrated into CI/CD and permission management workflows, would help developers identify privilege chains before deployment.

VII. Related Work

A. Serverless Security

Attacks in serverless. Multiple works discuss attacks via weak spots of serverless platforms or applications. Wang et al. [60] analyzed the serverless platforms and revealed that the placement vulnerabilities of Azure Function and the accounting flaws of Google Cloud Function. Yelam et al. [61] demonstrated a practical covert channel in AWS Lambda that can be leveraged to launch co-residence attacks. Xiong et al. [62] debuted the Warmonger attack, a novel attack vector that can trigger platform-wide denial-of-service by abusing shared egress IPs in serverless platforms. Shen et al. [63] performed a real-world internal Denial-of-Wallet attack on commodity serverless platforms to evaluate its severity. Although they only revealed various security issues in serverless platforms and applications, they mostly focus on traditional attack vectors or their serverless variants.

Defenses in serverless. There are multiple works on securing serverless environments. Datta et al. [64] presented Valve, a serverless platform that enables developers to exert complete fine-grained control of information flows in their applications by auditing network-layer flows and restricting function behavior. Sankaran et al. [65] proposed WILL.IAM, a workflow-aware access control model and reference monitor that can reduce the attack surface of the serverless application. Datta et al. [66] introduced ALASTOR, a provenance-based auditing framework that enables precise tracing of suspicious events in serverless applications. Similarly, Chen et al. [67] presented CLARION for auditing serverless containers. Jegan et al. [14] presented Kalium, an extensible security framework that leverages local function state and global application state to enforce control-flow integrity (CFI) in serverless applications. Polinsky et al. [68] identified opportunities for hardening serverless application policies and highlighting potential exfiltration channels. Rostamipoor et al. [69] presented LeakLess, combining in-memory encryption with a separate I/O module to safely transmit protected data between serverless functions and external hosts.

While prior works do excel in serverless defense mechanisms, they focused on securing or analyzing intra-application permissions, e.g., the ability of one function to invoke another or to access an S3 bucket, while deprioritizing the potential danger of how risky permissions can be chained across applications and accounts, due to dynamic resource sharing.

B. IAM Misconfiguration Detection

Prior research has extensively explored misconfiguration detection in cloud IAM systems, particularly within AWS. Shevrin et al. [38] modeled IAM transitions as finite-state machines to discover multi-step privilege escalation chains. Van Ede et al. [70] applied graph embedding and anomaly detection over policy graphs to identify over-privileged roles. Other tools, such as IAMGraph [15] and PMapper [43], construct privilege graphs to model user-role-resource relationships. Cloudsplaining [41], Red-Shadow [42], and AWS AccessAnalyzer [17] focus on flagging dangerous policies(e.g., wildcard use, known escalation patterns). IAMSpy [16] and the AWS Policy Simulator [44] use constraint solvers to reason about complex policy combinations.

While these tools effectively analyze privilege escalation within traditional cloud environments, they primarily focus on user-centric and role-centric analysis within single accounts. The key distinction in serverless environments is the prevalence of dynamic resource sharing mechanisms (e.g., Lambda Layers, container images) that enable automatic code composition across account boundaries. This creates novel cross-account privilege propagation paths that extend beyond traditional intra-account IAM analysis, as risky permissions can be systematically chained through shared resources to compromise multiple accounts.

VIII. Conclusion

In this paper, we reveal a fundamental mismatch between decentralized serverless applications and centralized permission management. This mismatch gives rise to an amplified attack surface, i.e., *risky permission chaining attacks*, where loosely scoped permissions can be chained vertically to compromise a single account, or propagated horizontally to compromise multiple accounts. To systematically analyze this new attack vector, we propose three general attack modalities and implement an automated analysis tool to detect such exploitable paths by matching against modality templates. To evaluate the impact, we apply our approach to AWS and Alibaba Cloud repositories. As a result, we have uncovered 28 vulnerable applications, including five new CVEs, six developer confirmations, and one security bounty. Finally, we provide several actionable suggestions to mitigate the risks.

Ethical Considerations

In this paper, we evaluate the security impact and identify risks introduced by chaining risky permissions in serverless applications. To align with ethical research standards, we adhere to two key principles: (1) the principle of minimizing harm, and (2) responsible disclosure. Specifically, we ensure that our research does not cause unintended real-world consequences and that any identified vulnerabilities are communicated securely and responsibly.

Environment Setup: All experiments are conducted in isolated environments that do not involve any other real users or deployed systems. This ensures that our evaluation does not interfere with operational serverless applications.

More specifically, for each cloud vendor we evaluate, we create multiple isolated customer accounts that are fully owned and managed by us. Within each account, we deploy serverless applications obtained from the serverless application repository, following official deployment guidelines. All intra-account experiments are confined strictly within a single self-owned account. For cross-account attack scenarios, we simulate the environment by creating multiple accounts ourselves; no experiments target or interact with any third-party or uninvolved tenants on the cloud platforms.

Responsible Disclosure: All discovered security risks have been disclosed to the relevant vendors or platform maintainers before the public release of our findings, to mitigate any potential abuse by malicious actors.

More specifically, for each potentially serverless application identified by our approach, we privately report the findings to the corresponding developers through appropriate disclosure channels. In this paper, we discuss only applications for which we have received formal acknowledgments (e.g., CVE assignment, bounty, or formal confirmation from the developers). For those applications with unconfirmed risks, we refrain from disclosing any technical details until we obtain confirmation from the developers.

Acknowledgment

The authors would like to thank all reviewers for their valuable comments. This work was partially supported by the National Natural Science Foundation of China (Key Program Grant No. 62232013), by the Foundation for Innovative Research Groups of the National Natural Science Foundation of China (Grant No. 62121001), by the Key Research and Development Program of Shaanxi (Program No. 2025CY-YBXM-066), and by the '111 Center' (B16037).

References

[1] DataDog, "The state of serverless," https://www.datadoghq.com/state-of-serverless/, 2025.
[2] G. V. Research, "Serverless computing market size," https://www.grandviewresearch.com/industry-analysis/serverless-computing-market-report, 2025.
[3] MarketsandMarkets, "The rise of serverless computing market: A $44.7 billion industry dominated by tech giants - aws (us) and microsoft (us)— marketsandmarkets," https://www.globenewswire.com/news-release/2024/10/04/2958384/0/en/The-Rise-of-Serverless-Computing-Market-A-44-7-billion-Industry-Dominated-by-Tech-Giants-AWS-US-and-Microsoft-US-MarketsandMarkets.html, 2025.
[4] Wiki, "Serverless computing," https://en.wikipedia.org/wiki/Serverless_computing, 2025.
[5] A. W. Services, "Coca-cola freestyle launches touchless fountain experience in 100 days using aws lambda," https://aws.amazon.com/solutions/case-studies/coca-cola-freestyle/, 2025.
[6] G. Cloud, "Telecom argentina speeds up technical incident resolution using google cloud tools," https://cloud.google.com/customers/telecom-argentina, 2025.
[7] Microsoft, ""tomoni®" accelerates digital innovation in power plants," https://www.microsoft.com/en/customers/story/1779541850037077027-mhi-azure-discrete-manufacturing-en-japan, 2025.
[8] A. Cloud, "Swire properties builds smart property platform on alibaba cloud function compute," https://www.alibabacloud.com/tc/customers/swireproperties, 2025.
[9] D. Sokolowski, P. Weisenburger, and G. Salvaneschi, "Automating serverless deployments for devops organizations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 57–69.
[10] G. Adzic and R. Chatley, "Serverless computing: economic and architectural impact," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 884–889.
[11] AWS, "Aws iam," https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html, 2025.
[12] A. Cloud, "Alibaba ram," https://www.alibabacloud.com/help/en/ram/product-overview/what-is-ram, 2023.
[13] OWSAP, "Owasp serverless top 10," https://owasp.org/www-project-serverless-top-10/, 2025.
[14] D. S. Jegan, L. Wang, S. Bhagat, and M. Swift, "Guarding serverless applications with kalium," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4087–4104.
[15] W. S. Labs, "Iamgraph," https://github.com/WithSecureLabs/IAMGraph, 2025.
[16] ——, "Iamspy," https://github.com/WithSecureLabs/IAMSpy, 2025.
[17] AWS, "Aws iam access analyzer," https://docs.aws.amazon.com/zh_cn/IAM/latest/UserGuide/what-is-access-analyzer.html/, 2025.
[18] ——, "Aws serverless application repository," https://serverlessrepo.aws.amazon.com/applications, 2025.
[19] A. Cloud, "Serverless application center," https://www.alibabacloud.com/help/en/functioncompute/fc-2-0/user-guide/overview-5, 2024.
[20] AWS, "Serverless on aws," https://aws.amazon.com/serverless/, 2025.
[21] Azure, "Azure functions," https://azure.microsoft.com/en-us/products/functions/, 2025.
[22] G. Cloud, "Serverless," https://cloud.google.com/serverless, 2025.
[23] I. Cloud, "Getting started with ibm cloud code engine," https://cloud.ibm.com/docs/codeengine?topic=codeengine-getting-started, 2025.
[24] A. Cloud, "Getting started with ibm cloud code engine," https://www.alibabacloud.com/en/product/function-compute, 2025.

[25] AWS, "Why aws serverless application repository?" https://aws.amazon.com/serverless/serverlessrepo/, 2025.
[26] I. Cloud, "Samples for using code engine," https://github.com/IBM/CodeEngine, 2025.
[27] Azure, "Azure samples," https://github.com/Azure-Samples/, 2025.
[28] Google, "Google samples," https://cloud.google.com/functions/docs/samples, 2025.
[29] AWS, "Aws deployment package," https://docs.aws.amazon.com/lambda/latest/dg/images-create.html, 2025.
[30] ——, "Aws lambda execution role," https://docs.aws.amazon.com/lambda/latest/dg/lambda-intro-execution-role.html, 2025.
[31] ——, "Chapter layers," https://docs.aws.amazon.com/lambda/latest/dg/chapter-layers.html, 2025.
[32] ——, "Aws ecr," https://aws.amazon.com/ecr/, 2025.
[33] ——, "Managed policies and inline policies," https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_managed-vs-inline.html, 2025.
[34] N. Frichette, "Lambda steal iam credentials," https://hackingthe.cloud/aws/exploitation/lambda-steal-iam-credentials/, 2025.
[35] Sysdig, "Serverless security risks," https://sysdig.com/learn-cloud-native/serverless-security-risks-and-best-practices/, 2025.
[36] Blcakhat, "Hacking serverless runtimes," https://www.blackhat.com/us-17/briefings/schedule/#hacking-serverless-runtimes-profiling-aws-lambda-azuref-unctions-and-more-6434, 2025.
[37] SNAS, "Attacking serverless servers reverse engineering the aws, azure, and gcp function runtimes," https://www.youtube.com/watch?v=MM5hWTZd_nQ, 2025.
[38] I. Shevrin and O. Margalit, "Detecting {Multi-Step}{IAM} attacks in {AWS} environments via model checking," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6025–6042.
[39] R. Security, "Aws privilege escalation methods," https://rhinosecuritylabs.com/aws/aws-privilege-escalation-methods-mitigation/, 2025.
[40] AWS, "Privilege escalation with iam," https://aws.amazon.com/cn/blogs/security/tag/privilege-escalation/, 2025.
[41] salesforce, "Cloudsplaining," https://github.com/salesforce/cloudsplaining, 2025.
[42] lightspin tech, "Red shadow," https://github.com/lightspin-tech/red-shadow, 2025.
[43] nccgroup, "Pmapper," https://github.com/nccgroup/PMapper/, 2025.
[44] AWS, "Aws policy simulator," https://docs.aws.amazon.com/zh_cn/IAM/latest/UserGuide/access_policies_testing-policies.html/, 2025.
[45] D. Reading, "Attack aws account via lambda function," https://www.darkreading.com/cloud-security/securing-serverless-attacking-an-aws-account-via-a-lambda-function, 2025.
[46] S. Framework, "Serverless framework applications," https://www.serverless.com/, 2025.
[47] Awesome, "Awesome serverless applications," https://project-awesome.org/pmuens/awesome-serverless, 2025.
[48] OpenFass, "Openfass function store," https://github.com/openfaas/store-functions, 2025.
[49] A. Cloud, "Alibaba security response center," https://security.alibaba.com/, 2025.
[50] Amazon, "Shared responsibility model," https://aws.amazon.com/compliance/shared-responsibility-model/, 2025.
[51] G. Cloud, "Shared responsibility in assured workloads," https://cloud.google.com/architecture/framework/security/shared-responsibility-shared-fate, 2025.
[52] Microsoft, "Shared responsibility in the cloud," https://learn.microsoft.com/en-us/azure/security/fundamentals/shared-responsibility, 2025.
[53] A. Cloud, "Shared security responsibility model," https://www.alibabacloud.com/help/en/well-architected/latest/security-responsibility-model, 2025.
[54] AWS, "Create a lambda function using a container image," https://docs.aws.amazon.com/lambda/latest/dg/images-create.html, 2025.
[55] ——, "Aws s3," https://aws.amazon.com/pm/serv-s3/, 2025.
[56] ——, "What is amazon eventbridge?" https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-what-is.html, 2025.
[57] ——, "Aws stepfunction," https://aws.amazon.com/step-functions/, 2025.
[58] ——, "What is aws secrets manager?" https://docs.aws.amazon.com/secretsmanager/latest/userguide/intro.html, 2025.
[59] ——, "What is cfnstacksets," https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/what-is-cfnstacksets.html, 2025.
[60] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX annual technical conference (USENIX ATC 18)*, 2018, pp. 133–146.
[61] A. Yelam, S. Subbareddy, K. Ganesan, S. Savage, and A. Mirian, "Coresident evil: Covert communication in the cloud with lambdas," in *Proceedings of the Web Conference 2021*, 2021, pp. 1005–1016.
[62] J. Xiong, M. Wei, Z. Lu, and Y. Liu, "Warmonger: Inflicting denial-of-service via serverless functions in the cloud," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 955–969.
[63] J. Shen, H. Zhang, Y. Geng, J. Li, J. Wang, and M. Xu, "Gringotts: Fast and accurate internal denial-of-wallet detection for serverless computing," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2627–2641.
[64] P. Datta, P. Kumar, T. Morris, M. Grace, A. Rahmati, and A. Bates, "Valve: Securing function workflows on serverless computing platforms," in *Proceedings of The Web Conference 2020*, 2020, pp. 939–950.
[65] A. Sankaran, P. Datta, and A. Bates, "Workflow integration alleviates identity and access management in serverless computing," in *Proceedings of the 36th Annual Computer Security Applications Conference*, 2020, pp. 496–509.
[66] P. Datta, I. Polinsky, M. A. Inam, A. Bates, and W. Enck, "{ALASTOR}: Reconstructing the provenance of serverless intrusions," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2443–2460.
[67] X. Chen, H. Irshad, Y. Chen, A. Gehani, and V. Yegneswaran, "{CLARION}: Sound and clear provenance tracking for microservice deployments," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3989–4006.
[68] I. Polinsky, P. Datta, A. Bates, and W. Enck, "Grasp: Hardening serverless applications through graph reachability analysis of security policies," in *Proceedings of the ACM Web Conference 2024*, 2024, pp. 1644–1655.
[69] R. Maryam, G. Seyedhamed, and P. Michalis, "Leakless: Selective data protection against memory leakage attacks for serverless platforms," in *32nd Annual Network and Distributed System Security Symposium (NDSS 25)*, 2025. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/leakless-selective-data-protection-against-memory-leakage-attacks-for-serverless-platforms/
[70] T. S. van Ede, "Comprehending security events: Context-based identification and explanation," 2023.

Appendix A
High-risk Permissions

In this appendix, we summarize high-risk permissions discussed in this paper in two tables, i.e., Table V for AWS and Table VI for Alibaba Cloud, respectively. These permissions are critical as they grant significant access or control over cloud resources and, if misused, can lead to privilege escalation, unauthorized resource access, or other security threats.

TABLE V: High-Risk Permissions in AWS

Platform	Single Account/Cross-account	Action	Semantics
AWS	Single Account	iam:CreateUser	Turns on a to-be-created user resource flag
		iam:UpdateUser	Updates the name attribute of the user
		iam:CreateLoginProfile	Adds the user to the attacker's credentials list
		iam:UpdateLoginProfile	Adds the user to the attacker's credentials list
		iam:PutUserPolicy	Updates user's inline policy to fully privileged (updated state)
		iam:DeleteUserPolicy	Removes user's inline policy (deleted state)
		iam:AttachUserPolicy	Adds a given managed policy to user's attached policies list
		iam:DetachUserPolicy	Removes a given managed policy from user's attached policies list
		iam:PutUserPermissionsBoundary	Updates user's permissions boundary policy
		iam:DeleteUserPermissionsBoundary	Removes user's permissions boundary policy
		iam:CreateGroup	Turns on a to-be-created group resource flag
		iam:UpdateGroup	Updates the name attribute of group
		iam:PutGroupPolicy	Updates group's inline policy to fully privileged (updated state)
		iam:DeleteGroupPolicy	Removes group's inline policy (deleted state)
		iam:AttachGroupPolicy	Adds a managed policy to group's attached policies list
		iam:DetachGroupPolicy	Removes a managed policy from group's attached policies list
		iam:AddUserToGroup	Adds the group to a user's groups list
		iam:RemoveUserFromGroup	Removes the group from a user's groups list
		iam:CreateRole	Turns on a to-be-created role resource flag
		iam:AssumeRole	Adds the role to the attacker's credentials list with a new session name
		iam:UpdateAssumeRolePolicy	Updates role's trust policy to fully privileged (updated state)
		iam:PutRolePolicy	Updates role's inline policy to fully privileged (updated state)
		iam:DeleteRolePolicy	Removes role's inline policy (deleted state)
		iam:AttachRolePolicy	Adds a managed policy to role's attached policies list
		iam:DetachRolePolicy	Removes a managed policy from role's attached policies list
		iam:PutRolePermissionsBoundary	Updates role's permissions boundary policy
		iam:DeleteRolePermissionsBoundary	Removes role's permissions boundary policy
		iam:CreatePolicy	Turns on a to-be-created policy resource flag
		iam:CreatePolicyVersion	Updates the policy to fully privileged (must have less than 5 versions)
		iam:DeletePolicyVersion	Deletes an arbitrary policy version (to have less than 5 versions)
		iam:SetDefaultPolicyVersion	Updates the default version of the policy
		iam:CreateInstanceProfile	Turns on a to-be-created instance profile resource flag
		iam:AddRoleToInstanceProfile	Updates instance profile's role
		iam:RemoveRoleFromInstanceProfile	Removes instance profile's role
		iam:CreateAccessKey	Adds the user's access key to the attacker's credentials list
		sts:AssumeRole	Grants permission to assume roles.
		organizations:UpdatePolicy	Updates the service control policy to fully privileged (updated state)
		organizations:AttachPolicy	Adds an SCP to account's or to organizational unit's SCP list
		organizations:DetachPolicy	Removes an SCP from account's or from organizational unit's SCP list
		lambda:InvokeFunction	Adds the function's execution role to the attacker's credentials list
		lambda:AddPermission	Updates function's resource policy to fully privileged (updated state)
		lambda:UpdateFunctionCode	Updates function's code to reveal role credential on execution
		lambda:CreateEventSourceMapping	Adds the EventSourceMapping's execution role to the credentials list
		lambda:UpdateEventSourceMapping	Adds the EventSourceMapping's execution role to the credentials list
		ec2:RunInstances	Turns on a to-be-created instance resource flag
		ec2-instance-ssh ("SSH into instance")	Adds the instance's role to the attacker's credentials list
		ec2:AssociateIamInstanceProfile	Updates instance's instance role in case it was empty
		ec2:DisassociateIamInstanceProfile	Removes instance's instance role
		ec2:ReplaceIamInstanceProfileAssociation	Updates instance's instance role in case it was already set
		s3:PutBucketPolicy	Updates bucket's resource policy to fully privileged (updated state)
		s3:DeleteBucketPolicy	Removes bucket's resource policy (deleted state)
		kms:PutKeyPolicy	Updates key's resource policy to fully privileged (updated state)
		kms:Decrypt	Must be allowed when accessing encrypted resources such as S3 buckets
		glue:CreateDevEndpoint	Adds the DevEndpoint's role to the attacker's credentials list
		glue:UpdateDevEndpoint	Updates the DevEndpoint's role
		cloudformation:UpdateStack	Adds the stack role to the attacker's credentials list
		cloudformation:CreateStack	Adds the stack role to the attacker's credentials list
		datapipeline:CreatePipeline	Turns on a to-be-created pipeline resource flag
		datapipeline:PutPipelineDefinition	Updates pipeline's role
		datapipeline:ActivatePipeline	Adds the pipeline's role to the attacker's credentials list
		iam:PassRole	An action needed for assigning roles, like Lambda execution or instance profiles
		lambda:CreateFunction	Turns on a to-be-created function resource flag
	Cross-account	lambda:UpdateFunctionConfiguration	Updates function's execution role (Lambda Layers)
		ecr:BatchGetImage	Allows retrieval of container images (ECR container images)
		ecr:GetDownloadUrlForLayer	Allows downloading image layers (ECR container images)
		s3:PutObject	Allows uploading objects to an S3 bucket (S3 buckets)
		s3:GetObject	Allows retrieving objects from an S3 bucket (S3 buckets)
		events:PutEvents	Allows putting events into an EventBridge event bus (EventBridge)
		states:CreateStateMachine	Allows creation of Step Functions state machines (Step Functions)
		states:PutResourcePolicy	Allows setting resource policy for Step Functions (Step Functions)
		states:StartExecution	Allows starting execution of a Step Functions state machine (Step Functions)
		secretsmanager:PutResourcePolicy	Allows attaching a resource policy to a secret for cross-account access (Secrets Manager)
		secretsmanager:PutSecretValue	Allows writing/updating secrets in Secrets Manager (Secrets Manager)
		secretsmanager:GetSecretValue	Allows reading secrets from Secrets Manager (Secrets Manager)
		cloudformation:CreateStackSet	Allows creation of CloudFormation StackSets (CloudFormation StackSets)

TABLE VI: High-Risk Permissions in Alibaba Cloud

Platform	Single Account/Cross-account	Action	Semantics
Alibaba Cloud	Single Account	ram:CreateUser	Turns on a to-be-created user resource flag
		ram:UpdateUser	Updates the name attribute of the user
		ram:CreateLoginProfile	Adds the user to the attacker's credential list
		ram:UpdateLoginProfile	Adds the user to the attacker's credential list
		ram:PutUserPolicy	Updates user's inline policy to fully privileged (updated state)
		ram:DeleteUserPolicy	Removes user's inline policy (deleted state)
		ram:AttachPolicyToUser	Adds a given managed policy to user's attached policies list
		ram:DetachPolicyFromUser	Removes a given managed policy from user's attached policies list
		ram:CreateGroup	Turns on a to-be-created group resource flag
		ram:UpdateGroup	Updates the name attribute of group
		ram:PutGroupPolicy	Updates group's inline policy to fully privileged (updated state)
		ram:DeleteGroupPolicy	Removes group's inline policy (deleted state)
		ram:AttachPolicyToGroup	Adds a managed policy to group's attached policies list
		ram:DetachPolicyFromGroup	Removes a managed policy from group's attached policies list
		ram:AddUserToGroup	Adds the group to a user's groups list
		ram:RemoveUserFromGroup	Removes the group from a user's groups list
		ram:CreateRole	Turns on a to-be-created role resource flag
		ram:UpdateRole	Updates the trust policy document of a RAM role
		ram:PutRolePolicy	Updates role's inline policy to fully privileged (updated state)
		ram:DeleteRolePolicy	Removes role's inline policy (deleted state)
		ram:AttachPolicyToRole	Adds a managed policy to role's attached policies list
		ram:DetachPolicyFromRole	Removes a managed policy from role's attached policies list
		ram:CreatePolicy	Turns on a to-be-created policy resource flag
		ram:CreatePolicyVersion	Updates the policy to fully privileged (must have less than 5 versions)
		ram:DeletePolicyVersion	Deletes an arbitrary policy version (to have less than 5 versions)
		ram:SetDefaultPolicyVersion	Updates the default version of the policy
		ram:CreateAccessKey	Adds the user's access key to the attacker's credentials list
		ram:AssumeRole	Grants permission to assume roles
		ram:PassRole	An action required for setting function execution or instance profile roles.
		sts:AssumeRole	Grants permission to assume roles.
		fc:InvokeFunction	Adds the function's execution ram role to the attacker's credential list
		fc:PutFunctionInvocationPermission	Updates function's resource policy to fully privileged (updated state)
		fc:UpdateFunctionCode	Updates function's code to reveal role credential on execution
		fc:CreateTrigger	Adds the Trigger's execution role to the credentials list
		fc:UpdateTrigger	Adds the Trigger's execution role to the credentials list
		fc:CreateFunction	Turns on a to-be-created function resource flag
		ecs:RunInstances	Turns on a to-be-created instance resource flag
		ecs:AttachInstanceRamRole	Updates instance's instance role in case it was empty
		ecs:DetachInstanceRamRole	Removes instance's instance role
		ecs:ReplaceInstanceRamRole	Updates instance's instance role in case it was already set
		oss:PutBucketPolicy	Updates bucket's resource policy to fully privileged (updated state)
		oss:DeleteBucketPolicy	Removes bucket's resource policy (deleted state)
		kms:PutPolicy	Updates key's resource policy to fully privileged (updated state)
		kms:Decrypt	Must be allowed when accessing encrypted resources such as OSS buckets
	Cross-account	fc:UpdateFunctionConfiguration	Updates function's execution role (Function Compute Layers)
		oss:PutObject	Allows uploading objects to OSS buckets (OSS buckets)
		oss:GetObject	Allows reading/downloading objects from OSS buckets (OSS buckets)
		eventbridge:PutEvents	Allows sending events to EventBridge event buses (EventBridge)
		ros:CreateStackGroup	Allows creating ROS StackGroups (Resource Orchestration Service)
		ros:CreateStackInstances	Allows deploying StackGroup instances (Resource Orchestration Service)
		ros:UpdateStackGroup	Allows updating StackGroups (Resource Orchestration Service)

Appendix B
Artifact Appendix

A. Description & Requirements

This appendix presents a comprehensive roadmap for setting up and evaluating the permission chaining attack of serverless apps provided by cloud vendors. This artifact mainly consists of two scanning tools designed for Amazon Web Service (AWS) serverless platforms and Alibaba Cloud Serverless platforms. Besides, we also provide our detailed scanning results for validation.

The tools enumerate all roles and their attached policies in the account, analyze serverless applications along with their associated roles, assess privilege escalation paths, simulate cross-account compromise scenarios, and generate detailed reports of confirmed attack paths, including function-based vectors and locations of sensitive permissions.

1) How to access: The artifact is hosted at: https://zenodo.org/records/16957393.

Artifact DOI: https://doi.org/10.5281/zenodo.16957393.

It contains:

- aws_escalate.py (Scanner for Amazon Web Services)
- aliyun_escalate.py (Scanner for Alibaba Cloud)
- requirements.txt (Python libraries for aliyun_escalate.py)
- fc2 (a Python library folder for aliyun_escalate.py)
- Example output files (aws_output.txt, aliyun_output.txt)
- LICENSE
- README.md

2) Hardware dependencies: Our artifact experiments are conducted on both serverless platforms provided by AWS and Alibaba Cloud. Accessing these cloud environments requires only standard laptops or desktops.

3) Software dependencies:

- Scanner for Amazon Web Services:
  The scanning tool for AWS serverless platform operates directly within the AWS CloudShell environment, an integrated terminal provided by the AWS cloud platform. To use this tool, you will need your AWS account credentials and access keys for AWS users or roles to authenticate and perform the required operations.
- Scanner for Alibaba Cloud:
  The scanning tool for the Aliyun serverless platform is executed locally in a Python environment, requiring Python version 3.10 or higher. It utilizes the fc2 library and other dependencies specified in the requirements.txt file. While the tool runs locally, it interacts with the Alibaba Cloud platform using access keys for Aliyun users or roles for authentication.

4) Benchmarks: For data validation, we also provide the repositories of the installed application, which can be installed and scanned as a benchmark to run our tool and validate the results. For the AWS environment, these applications originate from two primary sources: the AWS Serverless Application Repository and repositories from the AWS Samples organization on GitHub, which contains thousands of serverless apps, including dozens of vulnerable serverless applications, such as aws-deployment-framework and measure-cold-start, among others.

For the scanning tool targeting Alibaba Cloud, we use the Alibaba Serverless Application Center as the data source. It provides almost one hundred serverless applications, which contain several vulnerable serverless applications, including fc-llm-api and fc-stable-diffusion-plus.

B. Artifact Installation & Configuration

1) Scanner for AWS (aws_escalate.py):

- Sign up for an AWS account and log in to the AWS Management Console. Our AWS account username and password have been included in the 'Hardware and other requirements' field of the artifact submission page. (The AWS website is https://console.aws.amazon.com/)
- Deploy serverless applications. Open the Lambda page on the AWS platform, click "Applications" in the right sidebar, then click "Create application" on the left. On the page that opens, select "Serverless application" and choose an application template to create your serverless application. Taking measure-cold-start as an example, direct deployment from the AWS Serverless Application Repository may result in a runtime mismatch error, indicating incompatibility with the current AWS platform. The solution involves modifying the 'runtime' parameter in the application template to specify a supported runtime version, after which the application can be successfully redeployed.
- Upload aws_escalate.py file to the /home/cloudshell-user directory in AWS CloudShell.
- Update the 'credentials' file located in the /home/cloudshell-user/.aws directory with the following format:

```
[default]
aws_access_key_id = <AccessKeyID>
aws_secret_access_key = <SecretAccessKey>
```

  The user or role associated with the access key must have sufficient permissions to enumerate IAM roles, Lambda functions, and their associated policies. We have provided the actual values for AccessKeyID and SecretAccessKey in the 'Hardware and other requirements' field of the artifact submission page.

2) Scanner for Aliyun (aliyun_escalate.py):

- Sign up for an Aliyun account. (The Aliyun website is https://home.console.aliyun.com/)
- Deploy serverless applications. Open the Function Compute page on the Aliyun platform, click "Applications" in the right sidebar, then click "Create Application" on the left. On the page that opens, click 'Create Application from Template' and select an application template to create your serverless application. For instance, to deploy the fc-stable-diffusion application, first locate it in the Alibaba Serverless Application Center, click to access its deployment page, configure the required parameters, and proceed with deployment.

- Use the following command to install the required Python libraries within the local Python environment, instead of Aliyun CloudShell:

```
pip install -r requirements.txt
```

- Copy the folder 'fc2' to the corresponding location of the reference function library. Ensure you have installed all necessary Python libraries for running the scanner.
- Create aliyun_input.txt file with the appropriate values corresponding to your environment. The 'Hardware and other requirements' field on the artifact submission page contains the specific values we have provided. The content format of aliyun_input.txt is as follows:

```
<Access Key ID>
<Secret Access Key>
<Region ID>
<Account ID>
```

  Note: There must be a newline after '<Account ID>'.
- Place aliyun_input.txt in the same directory as aliyun_escalate.py.

C. Major Claims

- (C1): We characterize a direct privilege escalation attack in one account where over-privileged serverless functions can be abused to immediately obtain account administrative access. As depicted in Modality 1 of Subsection 2, Section C, Part IV of the paper. The functionality is tested in (E1).
- (C2): We characterize an indirect privilege escalation attack where compromised functions hijack critical functions by abusing over-privileged permissions to create new functions that inherit high-privilege roles. After that, the attacker can abuse the hijacked high-privileged functions to make privilege escalations similar to C1. As depicted in Modality 2 of Subsection 2, Section C, Part IV of the paper. The functionality is tested in (E2).
- (C3): We characterize a cross-account infection attack where compromised accounts abuse resource sharing mechanisms to spread malicious payloads to other accounts, enabling recursive privilege escalation across multiple account boundaries. As depicted in Modality 3 of Subsection 2, Section C, Part IV of the paper. The functionality is tested in (E3).

D. Evaluation

1) (E1): [Confirmed Modality 1] [2 compute-minutes (with pre-installed apps)]: Deploy a serverless application with privilege escalation capabilities for modality 1. The scanner will identify these elevated permissions and generate the corresponding attack paths. Since Alibaba Cloud Serverless does not have a modality 1 application, we have only conducted evaluations on the AWS platform.

[Preparation]

Deploy serverless applications on the AWS platform.

[Execution]

Run command 'python ./aws_escalate.py -p default > aws_output.txt' in AWS CloudShell.

[Results]

It will output the sensitive permissions related to Modality 1 for all roles and functions in the account, and ultimately provide privilege escalation paths based on Modality 1.

2) (E2): [Confirmed Modality 2] [2 compute-minutes (with pre-installed apps)]: Deploy multiple serverless applications with privilege escalation capabilities for modality 2. The scanner will identify these elevated permissions and generate the corresponding attack paths.

[Preparation]

Deploy serverless applications on both the AWS and Aliyun platforms.

[Execution]

- **AWS:** Run command 'python ./aws_escalate.py -p default > aws_output.txt' in AWS CloudShell.
- **Aliyun:** Run command 'python ./aliyun_escalate.py < aliyun_input.txt > aliyun_output.txt' in the directory containing aliyun_escalate.py.

[Results]

For AWS, it will identify and display the sensitive permissions related to Modality 1 and Modality 2 for all roles and functions in the account, and will ultimately present privilege escalation paths based on these modalities.

For Aliyun, it will display the sensitive permissions related to Modality 2 for all roles and functions in the account.

3) (E3): [Confirmed Modality 3] [2 compute-minutes (with pre-deployed environment)]: Configure an environment capable of cross-account contamination. The scanning tool will output cross-account attack paths.

[Preparation]

Repeat the steps from (E2).

[Execution]

Repeat the steps from (E2).

[Results]

For AWS, it will identify and display the sensitive permissions related to Modality 1, Modality 2 and Modality 3 for all roles and functions in the account, and will ultimately present privilege escalation paths based on these modalities.

For Aliyun, it will display the sensitive permissions related to Modality 3 for all roles and functions in the account.