ALPS: Automated Least-Privilege Enforcement for Securing Serverless Functions

Changhee Shin
Incheon National University
Republic of Korea
changhee9149@inu.ac.kr

Bom Kim
KAIST
Republic of Korea
spring@kaist.ac.kr

Seungsoo Lee
Incheon National University
Republic of Korea
seungsoo@inu.ac.kr

***Abstract*—Serverless computing is increasingly adopted for AI-driven workloads due to its automatic scaling and pay-as-you-go model. However, its function-based architecture creates significant security risks, including excessive privilege allocation and poor permission management. In this paper, we present** ALPS**, an automated framework for enforcing least privilege in serverless environments. Our system employs serverless-tailored static analysis to extract precise permission requirements from function code and a fine-tuned Large Language Model (LLM) to generate language- and vendor-specific security policies. It also performs real-time monitoring to block unauthorized access and adapt to policy or code changes, supporting heterogeneous cloud providers and programming languages. In an evaluation of 8,322 real-world functions across AWS, Google Cloud, and Azure,** ALPS **achieved 94.8% coverage for least-privilege extraction, improved security logic generation quality by 220% (BLEU), 124% (ChrF++) and 100% (ROUGE-2), and added minimum performance overhead. These results demonstrate that** ALPS **provides an effective, practical, and vendor-agnostic solution for securing serverless workloads.**

***Index Terms*—Serverless Computing, Function-as-a-Service, Least Privilege, Access Control**

I. Introduction

Serverless computing, a cloud model that enables developers to run applications without managing server infrastructure, is rapidly evolving with the rise of generative AI. AI inference workloads fluctuate significantly based on time and user demand, making them difficult to handle with traditional fixed infrastructure. Serverless architectures, with their automatic scaling and usage-based billing, address this challenge and have led to the emergence of Serverless Inference, which applies serverless principles to AI/ML model deployment. Major providers, including Hugging Face [1], AWS Lambda [2], Google Cloud Functions [3], Azure Functions [4], and Alibaba Function Compute [5], now support this approach. As a result, the global serverless computing market is projected to grow at a compound annual rate of 20.9%, from USD 121.6 billion in 2023 to USD 811 billion by 2033 [6].

However, those advancements introduce significant security challenges, particularly inadequate permission management and excessive authorization. Unlike traditional monolithic applications, serverless environments decompose business logic into numerous independent functions, each requiring specific IAM permissions. CheckPoint research indicates that 98% of serverless functions are over-privileged, with 16% posing critical security risks [7]. Developers often rely on wildcards (*) or overly broad managed policies for convenience, further exacerbating the problem [8].

Meanwhile, permission management issues persist after deployment. According to Orca Security (2023), 82% of organizations have IAM user credentials that haven't been used for at least 90 days, and 72% have unused IAM roles [9]. In the 2021 Sendtech data breach, an unrotated AWS access key created in 2015 allowed attackers to gain administrative access and steal personal data [10]. The 'set and forget' problem [11], where temporary privilege escalations are not revoked, is also widespread. In serverless environments, frequent function creation and deletion further increase the risk of indirect privilege abuse and make it impractical to maintain consistent security policies manually.

Several studies have attempted to address such challenges in serverless permission management. Static analysis approaches [12]–[19], identify security risks by examining function code or policy structures but cannot automatically derive the minimum permissions required, relying instead on manual policy creation. Log-based methods [20]–[22], generate policies from historical network activity or access logs but require lengthy data collection and fail to adapt to new or unforeseen traffic patterns. Runtime monitoring techniques [23]–[26] detect permission abuse or security breaches in real time but primarily function as post-mortem auditing tools, lacking immediate response capabilities. Overall, these approaches cannot automatically extract least privileges, identify excessive permissions, or detect mismatches between function code and assigned policies.

In this paper, we propose ALPS, an integrated framework that enforces least privilege in serverless environments. ALPS uses serverless-tailored static analysis to extract precise permission requirements from function code and a fine-tuned Large Language Model (LLM) to generate programming language- and vendor-specific security policies. It further monitors cloud service calls in real-time, blocking unauthorized access and adapting to policy or code changes. By supporting heterogeneous cloud vendors and runtime languages, ALPS provides a unified and automated approach to secure serverless function execution.

In the evaluation, ALPS demonstrates its effectiveness and practicality across diverse serverless environments. In a least-privilege extraction experiment involving 8,322 serverless

* Changhee Shin and Bom Kim contributed equally to this work. (This work was conducted while Bom Kim was with Incheon National University.) Seungsoo Lee (seungsoo@inu.ac.kr) is the corresponding author.

functions across AWS, Google Cloud, and Azure, our system achieved a coverage of 94.8%, confirming its applicability to real-world workloads. For security verification code generation, the fine-tuned LLM improved BLEU scores by 220%, ChrF++ scores by over 124% and ROUGE-2 scores by over 100% compared to the baseline, highlighting its ability to produce accurate, serverless-specific security logic. Additionally, the performance overhead remained low, with execution times increasing by an average of 3.9ms on AWS, 5.5ms on Google Cloud, and 7.2ms on Azure, while cost increases across all platforms were minimal. These results indicate that ALPS is both an effective and efficient solution for least-privilege management in production-grade serverless environments.

This paper makes the following contributions:

- We present the design and implementation of ALPS, a fully automated and vendor-agnostic framework for extracting and enforcing least privilege in serverless environments.
- We present an intelligent and adaptive verification code integration approach leveraging a fine-tuned LLM, enabling the system to monitor all access attempts and prevent privilege abuse at runtime without manual intervention.
- We present automated policy conversion that translates uniform security requirements into vendor-specific formats, ensuring consistent security across heterogeneous cloud environments.
- We evaluate ALPS across major cloud platforms and real-world serverless workloads, demonstrating high permission extraction accuracy, robust runtime enforcement, and minimal performance overhead.

II. Background and Problem Statements

A. Background

Serverless Function Invocation. Serverless computing [27]–[29] is an event-driven model where code executes at the function level, allowing developers to focus solely on business logic while cloud providers automate server provisioning, scaling, and maintenance through Function as a Service (FaaS). Popular platforms include AWS Lambda [2], Google Cloud Functions [3], Azure Functions [4], and Alibaba Function Compute [5]. As shown in Figure 1(A), user requests are routed via an API Gateway to the appropriate functions, which independently access cloud resources and return results. These functions are stateless and short-lived, terminating after execution, which enables a pay-as-you-go pricing model [30] where costs apply only to actual function runtime.

Functions are triggered via *API*, *Direct*, or *Event-driven* methods, each with specific security requirements. First, API Invocation is triggered via HTTP requests routed through API Gateways or load balancers, commonly used in user authentication, data retrieval, and real-time processing. Next, Direct Invocation explicitly calls another function, facilitating workflow composition and microservice inter-communication. Finally, Event-driven Invocation responds automatically to

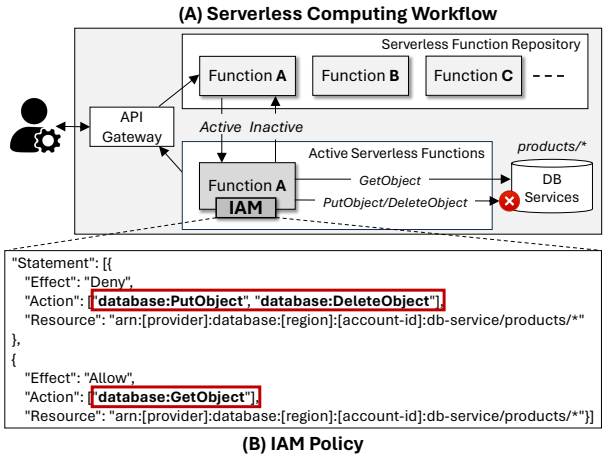

Fig. 1: The example of serverless computing workflow and general IAM-based access control.

state changes or external events (e.g., file uploads, database changes, message queues), suitable for batch processing, data pipelines, and real-time stream processing. Each method involves unique security implications, requiring fine-grained access controls to ensure secure interaction between functions and cloud resources.

Identity and Access Management (IAM). Identity and Access Management (IAM) [31] is an access control system responsible for managing permissions that enable serverless functions to interact with cloud resources. Due to their stateless nature, serverless functions cannot maintain permanent credentials. Consequently, each function invocation dynamically assigns temporary credentials based on predefined IAM policies, granting controlled access to resources. IAM enforces fine-grained access control by associating the entity requesting permissions (Identity), the defined permission rules (Action), and the targeted resource (Resource).

Figure 1(B) demonstrates IAM policy enforcement controlling resource access for a function within an AWS Lambda environment. When Function A attempts database access, IAM evaluates the assigned policies to determine permission grants or denials. In the depicted scenario, two IAM policies are applied: the first explicitly denies data insertion (`database:PutObject`) and deletion (`database:DeleteObject`), while the second explicitly allows data retrieval (`database:GetObject`). As a result, the function is permitted to read database contents upon invocation but is restricted from inserting or deleting data, illustrating real-time IAM policy enforcement.

B. Problem Statements

Enforcing the principle of least privilege in serverless environments presents significant challenges due to vendor-specific architectural differences, varying resource access requirements across individual functions, and dynamic changes in permission configurations during runtime. This section examines three critical challenges associated with enforcing effective least privilege in serverless function environments.

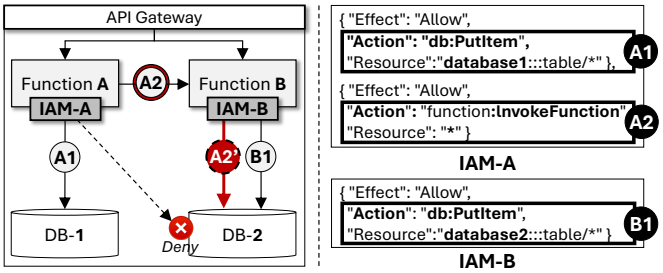

Fig. 2: The motivating example of unauthorized permission abuse through function invocation bypass (A2 → A2') caused by the excessive permission (A2).

C1: Risk of Function Privilege Abuse Due to Manual Policy Management. Developers often assign uniform or global policies to multiple functions without fully analyzing their specific resource requirements, and even when tailored policies are defined, they frequently rely on overly simplified structures or wildcard permissions that grant unrestricted access across functions and resources [13]. Also, managed policies provided by cloud vendors further aggravate this issue by including unnecessary permissions, as demonstrated by the vulnerability in the `AmazonGuardDutyFullAccess` policy, which enabled organization-wide compromise [32]. Although fine-grained permissions can be manually configured, the expertise and effort required make this approach impractical in large-scale serverless applications, thereby increasing the risk of privilege abuse.

Figure 2 illustrates how overly permissive policy allocation can expose serverless applications to indirect privilege escalation through function invocation chains or event-driven interactions. Function A has permissions limited strictly to DB-1, while Function B can access DB-2. Despite individual IAM policies effectively restricting direct resource access (A1, B1), excessive invocation permissions enable Function A (and thus the invoking user) to indirectly access unauthorized resources, such as DB-2 (A2'), by invoking Function B. Although each IAM policy remains individually compliant, excessive invocation permissions enable users to circumvent intended access constraints. Such issues occur when developers fail to implement fine-grained permission policies that restrict inter-function invocations to explicitly defined and necessary scenarios.

C2: Lack of Runtime Enforcement of the Principle of Least Privilege. In serverless environments, even if the principle of least privilege is correctly applied during initial deployment, privileges are often expanded or modified during operation without proper revocation, leading to the Set-and-Forget problem [11]. Existing permission verification mechanisms primarily rely on post-deployment analysis, which cannot prevent privilege abuse in real time [20], [24]. While cloud vendor monitoring tools can detect abnormal or unauthorized privilege usage through log analysis, they lack function-level blocking capabilities [33]–[35]. Similarly, policy-based approaches verify conditions at a logical level but do not enforce real-time, application-level checks to ensure resource access remains within permitted scopes. This absence of runtime verification allows privilege abuse to go undetected or unmitigated, significantly increasing the risk of security breaches.

C3: Lack of Consistency Across Cloud Providers in Serverless Environments. Each cloud provider uses a distinct IAM policy structure and permission management model, requiring developers to learn different syntax and configuration approaches for each platform [36], [37]. Serverless functions are also implemented in multiple programming languages, such as JavaScript, Python, and Go, further complicating the consistent enforcement of security policies across diverse language–platform combinations. Existing automated policy generation tools [38] are typically tightly coupled to specific vendors or languages, making them unsuitable for deployments across heterogeneous cloud providers. Consequently, maintaining uniform security policies across heterogeneous environments demands separate tools and processes for each platform, increasing management complexity and hindering effective security governance for serverless applications.

III. ALPS Design

This section outlines the design considerations that motivated the development of ALPS and presents its system architecture. Succinctly, our system enhances the security of serverless functions by automatically extracting least-privilege permissions through static analysis and verifying their correct operation under these permissions at runtime.

A. Design Considerations

1) Automatic Extraction of Fine-Grained Permissions for Enforcing Least Privilege. The system should accurately identify and extract the minimum permissions required for each serverless function based on its specific tasks. This requires automatically detecting cloud service calls and the corresponding resources accessed through function code analysis and deriving precise permission requirements using data flow inference. Furthermore, even in complex workflows involving direct invocations or event-driven function chaining, it should differentiate fine-grained permissions to ensure that each function is granted only the privileges necessary for its role, without manual developer intervention.

2) Real-Time Permission Monitoring and Abuse Prevention. The system should continuously ensure that functions operate within the minimum required privilege level after deployment. It should validate the legitimacy of all SDK service calls before execution using a dynamically generated whitelist derived from extracted minimum privileges and environment variables, immediately blocking any unauthorized access attempts. Furthermore, when function code or policies are modified, it should re-evaluate the function to update the whitelist, detect excessive privileges, and enforce appropriate execution controls.

3) Intelligent Integration Across Cloud Vendors and Programming Languages. The system should provide consistent authorization management across heterogeneous environ-

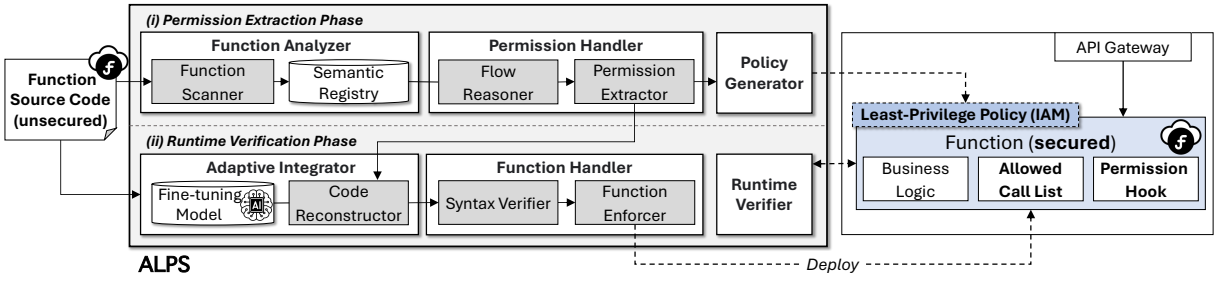

Fig. 3: Overall architecture and its workflow of ALPS with six key components: (*i*) function analyzer, (*ii*) permission handler, (*iii*) policy generator, (*iv*) adaptive integrator, (*v*) function handler and (*v*) runtime verifier. Additionally, our system includes two operational phases: permission extraction and runtime verification.

ments with varying policy grammars and code structures. This requires context-aware, intelligent code generation that goes beyond predefined template mappings by analyzing coding styles, variable naming patterns, and exception handling methods in real time to insert authorization verification logic without disrupting the existing program structure. Furthermore, the system should translate uniform security requirements into vendor-specific IAM policy grammars, enabling seamless management of heterogeneous cloud environments without additional configuration overhead for developers.

B. System Architecture and Workflow

To satisfy the design considerations outlined above, ALPS (depicted in Figure 3) comprises six core components: function analyzer, permission handler, policy generator, adaptive integrator, function handler, and runtime verifier. Our system accepts only the source code of serverless functions as input, without requiring any additional configuration files or metadata. The input functions represent typical serverless functions containing only standard business logic, without any security-specific modifications. Based on these inputs, the system operates in two stages: permission extraction phase and runtime verification phase, as described below.

First, the permission extraction phase automatically identifies the minimum permissions required for each serverless function and generates the corresponding policies. The *function analyzer* processes the source code, detects the programming language, and produces a structured code representation through static analysis, which is stored in the semantic registry as a knowledge base for subsequent analysis. Using this information, the *permission handler* detects cloud service calls and resource access paths, constructing a complete data flow that incorporates conditional branching, dynamic resource access, and environment variable references. Based on this flow, the permission extractor derives the minimal permission set (i.e., actions and resources) needed by the function, along with the resolved environment variable values, which are later used to create the allowed call list. Finally, the *policy generator* translates the extracted permissions into the appropriate IAM policy format for each cloud vendor.

Next, the runtime verification phase ensures that deployed functions operate strictly within their minimum required permissions while dynamically adapting to IAM policy changes after deployment. The *adaptive integrator* employs a fine-tuned Large Language Model (LLM), trained on a curated code integration dataset, to insert allowed call lists and permission monitoring hooks customized for each programming language and cloud provider. Guided by the code reconstructor, the LLM integrates extracted permissions and environment variables into the original function code, producing a secure, instrumented version. The *function handler* then validates the reconstructed code for syntactic correctness and functional equivalence before deploying it.

During execution, the *runtime verifier* inspects all SDK service calls in real-time. Each call is validated against a dynamically generated whitelist derived from environment variables, and any unauthorized access attempt is immediately blocked before execution. If function code changes, the updated code is reanalyzed in the permission extraction phase, and the whitelist is regenerated. Similarly, if an IAM policy change is detected, the system compares the updated policy against the whitelist and enforces execution within the scope of the whitelist, even if the policy grants excessive privileges.

IV. ALPS System Details

A. Automated Least-Privilege Extraction via Serverless-Tailored Static Analysis

In serverless environments, function-level source code is deployed directly to the cloud, granting developers full visibility and access. Each function has well-defined inputs, outputs, and logic, constraining its tasks and resource access. These properties allow static analysis to accurately identify execution flows and resource access patterns. Figure 4 illustrates how ALPS extracts least-privilege permissions.

Serverless-Tailored Static Analysis. Existing general-purpose static analysis tools, such as CodeQL [39] offer robust Abstract Syntax Tree (AST) generation and basic dataflow tracing across multiple languages. However, the complexity of cloud SDK services and the variability in call structures across vendors hinder their ability to accurately determine a function's precise permission requirements. To address this limitation, we extend the infrastructure of the CodeQL with a serverless-specific query framework tailored to accurately identify and trace cloud service calls.

(i) Language and Vendor Identification: The analysis begins with the function scanner automatically identifying the

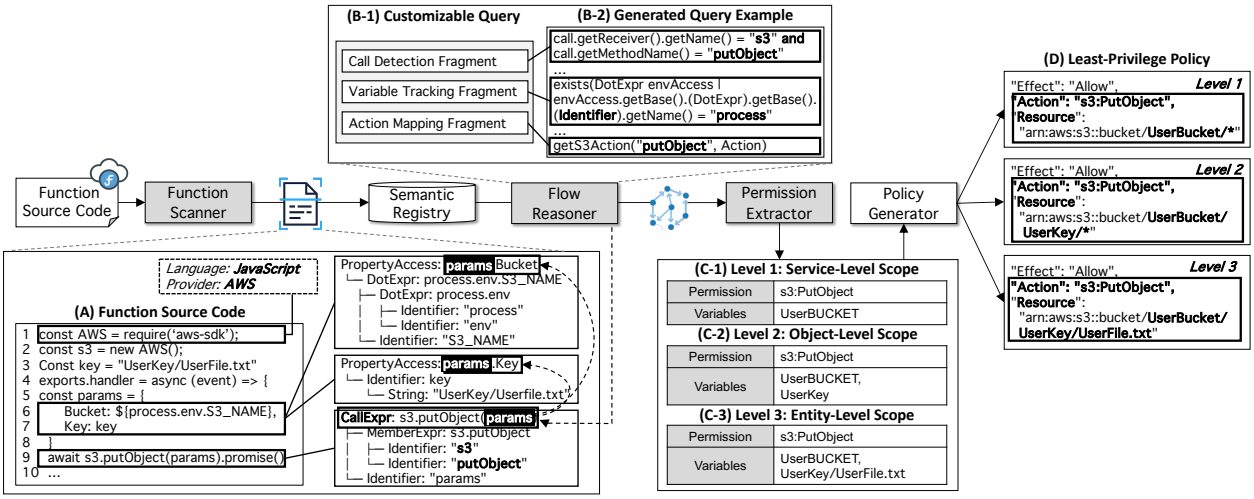

Fig. 4: The example of automatic extraction procedure of least-privileged policies and environment variables.

programming language and cloud vendor from the function source code ((A) in Figure 4). Language detection leverages file extensions, import statements, and syntax patterns, while vendor identification relies on analyzing SDK imports and client instantiation patterns. These language–vendor mappings form the basis for selecting the appropriate queries and rule sets for subsequent analysis.

(ii) AST-Based Structuring: Based on the identified language, CodeQL is employed to transform the function code ((A) in Figure 4) into an AST representation, enabling precise decomposition and structuring of cloud service call syntax. For example, instead of treating a cloud SDK call (`s3.putObject(params)`) as a single function call, it is decomposed into AST nodes that separately capture the service client (`s3`), method name (`putObject`), and parameters (`params`). Similarly, environment variable references (`process.env.S3_NAME`) and resource identifier accesses (`params.Bucket`, `params.Key`) are represented as individual nodes, allowing the tracking of dynamic resource reference patterns. The resulting semantic registry structures each component of a cloud service call into AST nodes and enables data flow tracking by automatically linking these nodes during query execution. This allows for more precise identification of authorization requirements in subsequent steps.

(iii) Automatic Query Building and Execution: To accurately identify and track cloud service calls, we develop a customizable query system ((B-1) in Figure 4) tailored to each language–vendor combination. This system comprises three main query fragments. The Call Detection Fragment detects service invocations ranging from direct cloud SDK calls to indirect calls made through wrapper functions or utility modules. The Variable Tracking Fragment traces the data flow of variables relevant to authorization decisions, including environment variable references (`process.env.BUCKET_NAME`), configuration file loading, and dynamically generated resource identifiers. Finally, the Action Mapping Fragment defines mapping rules that translate identified SDK method calls into specific IAM actions for the corresponding cloud vendor.

Based on the established query structure, the flow reasoner automatically generates the analysis queries optimized for the identified language–vendor combination (B-2). For example, for the AWS–JavaScript combination, it generates a query that detects the `putObject(UserBUCKET, UserKEY, putParams)` call and traces the data flow of the `UserBUCKET` and `UserKEY` variables. When executed on the semantic registry, the flow reasoner derives cloud service call points and their associated data flows. Through this process, it establishes the connection between the `putObject` call and the `UserBUCKET` and `UserKEY` variables, identifies dynamic resource reference patterns (e.g., environment variables), and determines all resources and actions accessible to the function during execution. This analysis forms the basis for the subsequent least-privilege extraction step.

Least Privilege Extraction. The permission extractor determines the minimum permissions required by a function based on the data flow derived by the flow reasoner. For each identified cloud service call point, it applies the Action Mapping Fragment rules to map method calls (`putObject`) to their corresponding IAM actions (`s3:PutObject`) and associates variables (`UserBUCKET`, `UserKEY`) with the corresponding bucket and object resources. For calls within conditional branches or loops, it analyzes all possible execution paths to ensure comprehensive permission coverage.

Based on the extracted permissions, the system provides three granular levels of access: (i) *Service-Level Scope* (C-1 in Figure 4), which grants only service-specific action permissions; (ii) *Object-Level Scope* (C-2), which additionally specifies key resource identifiers; and (iii) *Entity-Level Scope* (C-3), which produces the most fine-grained permissions, encompassing all relevant variables and execution paths. This approach minimizes the use of wildcards while allowing users to select the appropriate permission level based on operational and management requirements. Runtime enforcement, however, adheres to the most granular level (C-3) following the principle of least privilege.

Finally, the policy generator automatically translates the ex-

tracted permissions into vendor-specific IAM policy formats, including AWS, Google Cloud, and Azure, producing ready-to-deploy minimum privilege policies (D in Figure 4).

B. Real-Time Monitoring and Verification of Permission Abuse

ALPS implements a mechanism for intercepting and verifying all cloud service calls made by a deployed serverless function in real-time. This verification is achieved through permission hooks embedded in the function code, enabling continuous monitoring of all cloud service calls during execution and automatically accommodating configuration changes that occur after deployment. These hooks operate based on an allowed call list of permission–resource pairs, ensuring that it remains non-intrusive to the business logic of the function. The process for generating the allowlist and integrating permission hooks into the function code is described in Section IV-C.

Algorithm 1 Hierarchical Verification for Service Calls.

```
Input: Service Calls Event E
Output: 0(Allow) or 1(Deny)
action ← get_current_action(E)
service ← get_current_service(E)
if requires_resource_extraction(E) then
    resource ← extract_resource_name(E, service)
else
    resource ← default_resource_identifier(E)
allowlist ← get_allowed_calls()
is_allowed ← FALSE
if service ∈ allowlist then
    allowed_resource ← allowlist[service]
    if resource ∈ allow_resource then
        allowed_action ← allowed_resource[resource]
        if action ∈ allowed_action then
            is_allowed ← TRUE
if is_allowed == FALSE then
    return 1
return 0
```

The verification process begins by intercepting every service request through hooks in the core service mechanisms of the cloud SDK. For example, in the Azure SDK, the request of the Cosmos DB client method is intercepted to capture database calls before execution. From each call, the service, operation, and parameters are extracted, and the target resource is dynamically resolved. For Cosmos DB, database and container names are analyzed, and if resource identifiers are environment variables, their values are retrieved at runtime. The identified service–resource pair and operation are then checked against the allowlist of the function. Unauthorized requests raise an exception and are blocked immediately. These steps are summarized in Algorithm 1.

Automated Handling of Post-Deployment Configuration Changes. In serverless environments, various configuration changes can occur after deployment. ALPS automatically adapts to these changes, ensuring continuous compliance with the principle of least privilege. When the source code of a deployed function is updated, the updated function code is reintroduced into the static analysis stage, where new permission sets and environment variables are derived, and the allowlist is regenerated.

TABLE I: The summary of the datasets for LLM fine-tuning (Fn = Function, FT = Fine-tuning Set). Origin function templates are sourced from the *Wonderless* dataset [40].

	AWS			**Google Cloud**			**Azure**	
	JS	Py	Go	JS	Py	Go	JS	Py
Function	5,475	889	39	9	21	16	31	22
Permission	680			676			660	
Curated Fn	30	14	24	9	10	9	15	10
Refined FT	20,000	8,940	4,130	6,100	6,800	5,500	9,300	6,700

When an IAM policy is modified, our system determines whether excessive permissions have been introduced. Even if additional permissions are granted, our system enforces execution strictly within the previously extracted least-privilege scope. By comparing the permissions defined in the new policy with the current allowlist, ALPS identifies over-permissive resources or actions and restricts function execution.

C. LLM-Based Adaptive Code Integration

To ensure consistent authorization verification across diverse cloud vendors and programming languages in serverless environments, ALPS introduces a fine-tuned, LLM-based adaptive code integration mechanism. Existing general-purpose code generation models often fail to capture the specialized authorization verification requirements of serverless functions or account for differences in SDK architectures across cloud providers. In addition, template-based approaches are limited in their ability to seamlessly integrate authorization logic into function code. To address these gaps, we fine-tuned an LLM on a domain-specific dataset for serverless authorization management, enabling it to learn the unique characteristics of each environment and perform context-aware code integration.

Dataset Construction. The fine-tuning dataset was built using serverless functions from the *Wonderless* dataset [40] and permission policy examples collected from GitHub and official documentation (Table I). To ensure high-quality training data, we applied rigorous filtering criteria, including the removal of inactive projects (those with less than one year of activity), official community examples, duplicate code, and empty functions. From approximately 6,500 functions gathered across three cloud vendors, we excluded Azure functions in Golang due to the lack of suitable examples. From the remaining combinations, about 120 functions were selected for their representativeness and diversity, emphasizing common resource access patterns and function structures. Additionally, around 2,000 permission policy examples were evenly collected from AWS, Google Cloud, and Azure. The selected functions were curated and augmented using semantic-preserving transformations [41] to ensure clear function-permission mappings while generating diverse training scenarios. This process yielded a high-quality dataset tailored to serverless permission management. The dataset was split into training and test sets (including validation) at an 8:2 ratio for reliable model evaluation.

LLM-Based Permission Verification Code Generation. As shown in Figure 5, the Base LLM is adapted into a Fine-Tuned LLM via transfer learning that incorporates structured

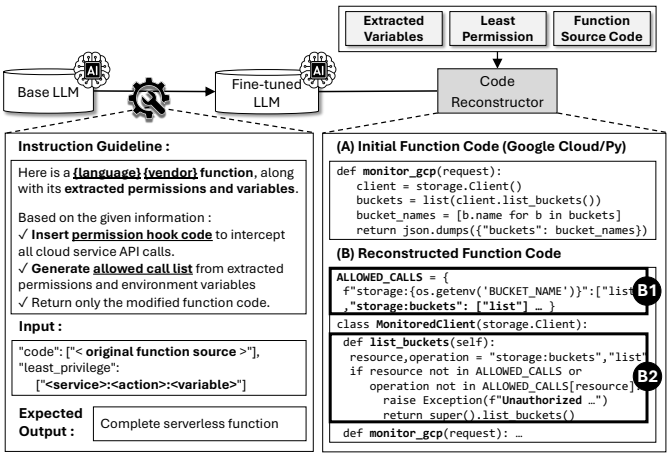

Fig. 5: Instruction guideline for LLM fine-tuning and the example for automatic function restructuring by the code reconstructor.

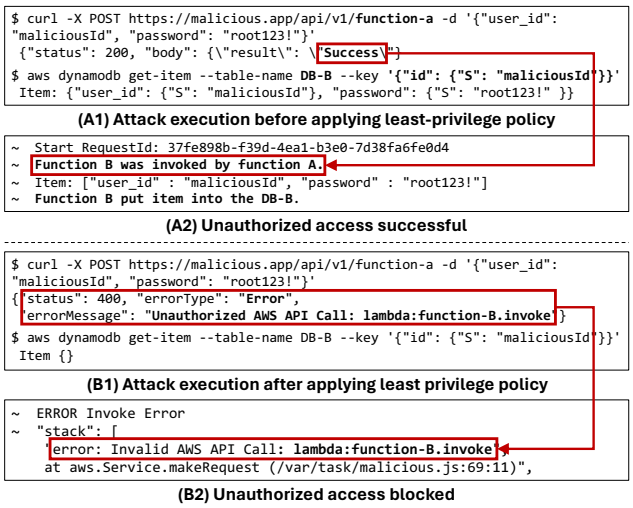

Fig. 6: The comparison of the permission abuse results with and without least privilege policies generated by ALPS.

context information, enabling precise serverless permission verification code generation. The Code Reconstructor uses three inputs: (i) variables from static analysis, (ii) the least-privilege permission set, and (iii) the original function code. The model is trained to generate vendor- and language-specific functions, insert permission hooks, and build environment-aware allowlists, producing fully instrumented functions.

The trained model automatically integrates these components. In Figure 5, the original function (A) only instantiated a Google Cloud Storage client, whereas the reconfigured function (B) includes permission verification logic. First, the model dynamically generates allowlists by combining environment variables with the derived permission set (B1). Second, it injects language-specific hooks: Python uses wrapper classes, JavaScript redefines SDK methods at runtime, and Go inserts explicit verification calls (B2). This approach enables automated, environment-optimized authorization enforcement with minimal developer intervention.

V. Evaluation

A. Implementation

We implemented a prototype of ALPS using CodeQL [39] and Python, comprising approximately 50,000 lines of code (primarily CodeQL queries, with ∼200 lines of core logic in Python), to evaluate its feasibility and effectiveness. The AST-based static analysis engine extracts resources, actions, and conditions from cloud service calls and infers the required permissions, which are then normalized into a unified representation compatible with AWS IAM [42], Google Cloud IAM [43], and Azure RBAC [44]. Furthermore, we fine-tuned the DeepSeek 6.7B model [45] to automatically generate runtime security verification logic from function code and least-privilege specifications, incorporating platform- and language-specific characteristics.

B. Evaluation Environments

The experiments were performed on a high-performance server equipped with an NVIDIA Hopper H100 80GB PCIe GPU, an AMD EPYC 9224 CPU, 256GB RAM, and a 2TB SSD. We tested AWS Lambda [2], Google Cloud Functions [3], and Azure Functions [4] with Node.js, Python, and Go runtimes. The functions were collected from Wonderless [40], a benchmark dataset containing real-world serverless functions that span heterogeneous vendors and runtimes, and exhibit key serverless characteristics such as event chaining, database access, and external service calls. This setup enabled a comprehensive evaluation of the capability of our system to extract least privilege, generate security logic, and verify and enforce policies under realistic conditions.

C. Functional Correctness

This evaluation examines the effectiveness of ALPS in enforcing least-privilege policies to prevent permission abuse. Figure 6 compares the abuse outcomes before and after applying least-privilege policies. The experiment simulates an AWS Lambda environment where Function A is erroneously granted excessive permissions, including the ability to invoke other functions using wildcards, despite requiring access only to DB-A, as illustrated in Figure II-B. We constructed a permission abuse scenario in which Function A exploits these unnecessary privileges to invoke Function B and insert malicious data into DB-B, thereby assessing the defensive capability of ALPS.

While Function A successfully inserted data into DB-B by invoking Function B due to excessive permissions, as shown in Figure 6 (A1-2), this attempt was promptly blocked after enforcing the least-privilege policy. ALPS analyzed the function's code, identified that it required only DB-A access permissions, and generated a policy that removed the unnecessary Lambda invocation privileges. The log in Figure 6 (B1-2) further confirms the invocation failure and shows that the attempt to insert data into DB-B was blocked. These results demonstrate that our system accurately identifies actual resource access patterns through serverless-specific static analysis and generates least-privilege policies that effectively eliminate unnecessary permissions.

TABLE II: Statistics of detection results per cloud platform and language. Origin Function templates are sourced from the *Wonderless* datasets [40].

	AWS			Google Cloud			Azure		Total
	JS	Py	Go	JS	Py	Go	JS	Py	
Function	2512	1208	807	784	802	753	852	604	8322
Detected	2393	1140	779	759	777	729	827	587	7891
Undetected	119	68	28	25	25	24	25	17	331

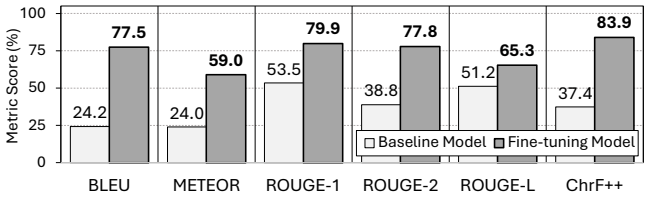

Fig. 7: The summary of the fine-tuning LLMs performance with metrics: BLEU, METEOR, ROUGE-1, ROUGE-2, ROUGE-L, ChrF++.

D. Performance

Function Coverage. We measure the coverage of permission extraction by ALPS across real serverless workloads to assess the success rate of least-privilege extraction for diverse functions. Specifically, it aims to determine whether our system can achieve complete extraction at the `entity-level` scope, the most granular of the three permission granularity levels proposed. As shown in Table II, unlike the fine-tuning dataset, the evaluation dataset was constructed by collecting all functions containing cloud service calls from the *Wonderless* dataset [40] and removing duplicates and empty templates. This process yielded approximately 8,300 unique functions spanning three major cloud platforms (AWS, Google Cloud, and Azure) and three programming languages (JavaScript, Python, and Go).

As a result, ALPS successfully extracted least-privilege policies for 7,891 functions, achieving 94.8% coverage. Google Cloud had the highest success rate at 96.8%, followed by AWS at 95.2% and Azure at 90.3%. Across languages, JavaScript, Python, and Go achieved 95.9%, 95.8%, and 90.3% respectively, demonstrating consistent performance. The 331 functions (5.2%) that failed entity-level extraction were mainly due to two static analysis limitations: dynamically determined resource identifiers (e.g., from service responses or user inputs) and service calls that require wildcard permissions (e.g., S3 ListBucket, Lambda ListFunctions). In these cases, only service-level permissions could be generated. These results confirm the broad applicability, vendor compatibility, and effectiveness of our system in extracting least-privilege policies across diverse serverless workloads.

Fine-tuning LLM Performance. To quantitatively evaluate the accuracy of verification logic generation in serverless environments, we compared the base model (DeepSeek Coder 6.7B Instruct [45]) with its fine-tuned version. For evaluation, we used a test set randomly sampled from 20% of the dataset defined in Table I. Each model generated function code with embedded verification logic based on the input function code and least-privilege specifications. Model performance was assessed using BLEU [1] [46], METEOR [2] [47], ROUGE (ROUGE-1, ROUGE-2, ROUGE-L) [3] [48] and ChrF++ [4] [49].

As summarized in Figure 7, the fine-tuned model achieved substantial performance improvements across all metrics. The BLEU score increased by approximately 220% (24.2 to 77.5), ROUGE-2 by over 100% (38.8 to 77.8), and ChrF++ by 124% (37.4 to 83.9), with consistent improvements observed in other metrics. These results demonstrate that the proposed approach can reliably generate high-quality verification logic that aligns with the complex permission requirements of serverless platforms and validate the effectiveness of fine-tuning in this context.

TABLE III: Cost comparison by cloud provider before and after applying ALPS ($\times 10^7$ USD)

	ALPS	1 Call	2 Calls	3 Calls	4 Calls	5 Calls
AWS	without	0.56	1.12	2.10	2.70	5.80
	with	0.59	1.31	2.11	2.80	5.86
Google Cloud	without	7.69	27.64	30.26	57.21	69.07
	with	8.28	29.23	30.74	59.39	70.20
Azure	without	0.90	1.23	1.90	2.92	4.05
	with	0.97	1.32	2.07	3.05	4.32

Performance Overheads. This experiment quantitatively evaluates the impact of verification logic, inserted through code reconstruction by our system, on the execution time and cost of serverless functions. The functions were categorized by the number of service calls per function across AWS Lambda [2], Google Cloud Functions [3], and Azure Functions [4]. Each function was executed 100 times in a warm-start environment with identical regions and resource configurations. As shown in Figure 8, AWS Lambda exhibited the lowest overhead (1.7 ms for one service call, 2.7 ms for five), followed by Google Cloud (2.7 ms and 5.2 ms) and Azure (3.4 ms and 13.3 ms). Average overheads were 3.9 ms, 5.5 ms, and 7.2 ms, respectively, all remaining below 10% across platforms.

Based on the measured execution times, we calculated the cost increase rates for each cloud platform using their respective pricing models. As shown in Table III, execution costs (scaled to $\times 10^7$ USD) increased on average by 5.5% for AWS, 4.1% for Google Cloud, and 7.0% for Azure. AWS showed the greatest variation, ranging from 0.5% to 16.9%, due to the interaction between its memory-based pricing and function complexity. Google Cloud showed relatively consistent cost increases driven by its cumulative vCPU and memory pricing. Azure exhibited steady cost increases proportional to overhead under its execution time pricing model.

[1]BLEU measures grammatical and syntactic similarity to the ground truth

[2]METEOR evaluates semantic alignment and word-level precision with synonym matching

[3]ROUGE evaluates semantic coverage of key permission verification components, including resources, action mappings, and conditional statements.

[4]ChrF++ measures character-level similarity for code structure accuracy

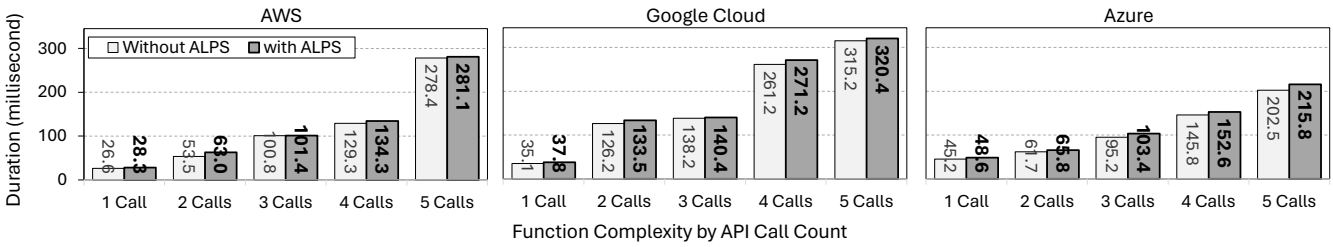

Fig. 8: The results of the duration with and without ALPS under different numbers of service calls in serverless functions across AWS, Google Cloud, and Azure.

Consequently, the performance and cost overhead introduced by our system remain within single-digit percentages, which is acceptable given the robust least-privilege security it provides. These results confirm that the approach offers a practical balance between security and efficiency in real-world serverless environments.

VI. Related Work

Serverless Permission Management and Flow Control. Recent research on serverless environments has focused on managing permissions and controlling information flow to enhance security, though most depend on specific languages or platforms [12], [13], [16], [18], [20]. Gupta et al. [12] proposed a policy-based compliance verification tool for AWS and Google Cloud that validates whether serverless applications adhere to developer-specified policies. However, it fundamentally assumes developers have already manually configured all permissions according to the principle of least privilege, requiring them to examine the Application Dataflow Graph (ADG) and write policies using complex Datalog syntax for each edge, thereby placing substantial manual burden on developers. In log-based approaches, D'Souza et al. [20] proposed a tool that automatically reduces AWS IAM policies through access log analysis. However, it requires data collection periods and is dependent on logs. Polinsky et al. [13] identified potential attack paths through graph-based policy analysis for serverless, yet they did not provide any countermeasures. Alpernas et al. [16] proposed dynamic information flow control for serverless, but it is limited to JavaScript and requires modifying function code. Datta et al. [18] proposed information flow control for function workflows through network-level taint tracking, but it is specialized for the OpenFaaS only, which limits its generality.

Unlike previous works, ALPS offers a vendor-agnostic framework that automatically extracts least privilege permissions through serverless static analysis, supporting multiple programming languages and cloud platforms. It performs three levels of granular permission extraction at the service, object, and entity levels across AWS, Google Cloud, and Azure, automating permission management by generating deployable least privilege policies directly from function code.

Runtime Monitoring and Dynamic Security Enforcement. In serverless environments, runtime security research mainly focuses on post-incident detection and auditing. [22]–[26] Shin et al. [23] proposed a dynamic security framework that detects and blocks privilege abuse and chain function call attacks in real time in serverless environments, but users must specify resources in advance for verification. Shen et al. [24] developed a performance monitoring system that detects internal Denial-of-Wallet attacks in serverless computing. However, it only provides attack detection and does not actually block attacks, which allows them to continue after detection. Alpernas et al. [25] proposed a monitoring and debugging system that detects runtime property violations in serverless applications. However, it only records violations in logs after the fact without performing real-time blocking. Additionally, Datta et al. [26] proposed provenance-based auditing for attack path reconstruction, and Satapathy et al. [22] proposed distributed provenance tracking; however, both focus solely on post-attack analysis. Meanwhile, LLM-based approaches for security automation are still in the early stages in serverless environments [50], [51]. These are limited to Wen et al. [50]'s work on detecting errors in AWS configuration files and Arun et al. [51]'s exploratory research on component generation.

While existing approaches are reactive, detecting security violations only after they occur, ALPS proactively prevents such violations in real time, blocks unauthorized service calls, and continuously enforces the principle of least privilege by automatically reevaluating code and policy changes using fine-tuned LLM-based adaptive code integration.

VII. Conclusion

This paper presents ALPS, a framework that integrates large language models with AST-based static analysis to automatically extract least privileges and insert dynamic verification logic in serverless environments. Unlike prior graph- or log-based methods, it provides continuous enforcement of the principle of least privilege while maintaining minimal runtime and cost overhead. Experiments demonstrate high accuracy in privilege extraction, effective verification logic generation, and strong compatibility across major cloud providers and programming languages. By offering a practical, lightweight solution for security automation, this work provides a foundation for advancing serverless security. The authors have made their code publicly accessible at [52] and their data at [53].

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) funded by the Korean Government (MSIT) under Grant RS-2025-16069415.

REFERENCES

[1] Hugging face Serverless Inference API, https://huggingface.co/learn/cookbook/en/enterprise_hub_serverless_inference_api, 2025.

[2] AWS Lambda, https://aws.amazon.com/pm/lambda/?nc2=h_mo-lang, 2025.

[3] Google Cloud Function, https://cloud.google.com/functions, 2025.

[4] Microsoft Azure Functions, https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview, 2025.

[5] Alibaba Cloud Function Compute, https://www.alibabacloud.com/en/product/function-compute?_p_lc=1, 2025.

[6] 2024 Serverless Computing Market Report, https://market.us/report/serverless-computing-market/, 2024.

[7] What is serverless security? , https://www.checkpoint.com/cyber-hub/cloud-security/what-is-serverless-security/, 2025.

[8] sysdig 2024 Cloud-Native Security and Usage Report, https://2631050.fs1.hubspotusercontent-na1.net/hubfs/2631050/Sysdig%202024-report-cloud-native-security-and-usage.pdf, 2025.

[9] Orca's 2023 State of Cloud Security Report, https://orca.security/wp-content/uploads/2024/02/2024-State-of-Cloud-Security-Report.pdf, 2025.

[10] PERSONAL DATA PROTECTION COMMISSION, https://www.pdpc.gov.sg/-/media/files/pdpc/pdf-files/commissions-decisions/decision---sendtech-pte-ltd---220721.pdf, 2025.

[11] "Barracuda networks, identity access and management checklist," https://assets.barracuda.com/assets/docs/dms/checklist_Identity_and_access_management_ENG.pdf, 2025.

[12] P. Gupta, A. Moghimi, D. Sisodraker, M. Shahrad, and A. Mehta, "Growlithe: A developer-centric compliance tool for serverless applications," in 2025 IEEE Symposium on Security and Privacy (SP). IEEE, 2025, pp. 3161–3179.

[13] 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.

[14] R. Shimizu and H. Kanuka, "Test-based least privilege discovery on cloud infrastructure as code," in 2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, 2020, pp. 1–8.

[15] P. Gill, W. Dietl, and M. Tripunitara, "Least-privilege calls to amazon web services," IEEE Transactions on Dependable and Secure Computing, vol. 20, no. 3, pp. 2085–2096, 2022.

[16] K. Alpernas, C. Flanagan, S. Fouladi, L. Ryzhyk, M. Sagiv, T. Schmitz, and K. Winstein, "Secure serverless computing using dynamic information flow control," Proceedings of the ACM on Programming Languages, vol. 2, no. OOPSLA, pp. 1–26, 2018.

[17] 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.

[18] 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.

[19] Y. Gu, X. Tan, Y. Zhang, S. Gao, and M. Yang, "Epscan: Automated detection of excessive rbac permissions in kubernetes applications," in 2025 IEEE Symposium on Security and Privacy (SP). IEEE, 2025, pp. 3199–3217.

[20] L. D'Antoni, S. Ding, A. Goel, M. Ramesh, N. Rungta, and C. Sung, "Automatically reducing privilege for access control policies," Proceedings of the ACM on Programming Languages, vol. 8, no. OOPSLA2, pp. 763–790, 2024.

[21] S. Xu, Q. Zhou, H. Huang, X. Jia, H. Du, Y. Chen, and Y. Xie, "Log2policy: An approach to generate fine-grained access control rules for microservices from scratch," in Proceedings of the 39th Annual Computer Security Applications Conference, 2023, pp. 229–240.

[22] U. Satapathy, R. Thakur, S. Chattopadhyay, and S. Chakraborty, "Disprotrack: Distributed provenance tracking over serverless applications," in IEEE INFOCOM 2023-IEEE Conference on Computer Communications. IEEE, 2023, pp. 1–10.

[23] C. Shin, B. Kim, and S. Lee, "Bambda: A real-time verification framework for serverless computing," IEEE Access, 2025.

[24] 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.

[25] K. Alpernas, A. Panda, L. Ryzhyk, and M. Sagiv, "Cloud-scale runtime verification of serverless applications," in Proceedings of the ACM Symposium on Cloud Computing, 2021, pp. 92–107.

[26] 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.

[27] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," ACM Computing Surveys (CSUR), vol. 54, no. 10s, pp. 1–34, 2022.

[28] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: a survey of opportunities, challenges, and applications," ACM Computing Surveys, vol. 54, no. 11s, pp. 1–32, 2022.

[29] H. B. Hassan, S. A. Barakat, and Q. I. Sarhan, "Survey on serverless computing," Journal of Cloud Computing, vol. 10, no. 1, p. 39, 2021.

[30] Serverless Framework Pricing, https://www.serverless.com/pricing, 2025.

[31] C. Singh, R. Thakkar, and J. Warraich, "Iam identity access management—importance in maintaining security systems within organizations," European Journal of Engineering and Technology Research, vol. 8, no. 4, pp. 30–38, 2023.

[32] Out of Sight, Beneath the Surface: Exploring Delegated Admin Risks in AWS Organizations, https://cymulate.com/blog/aws-delegated-admin-org-takeover/, 2025.

[33] AWS IAM Access Analyzer , https://aws.amazon.com/iam/access-analyzer/?nc1=h_ls, 2025.

[34] Policy Analyzer for allow policies, https://cloud.google.com/policy-intelligence/docs/policy-analyzer-overview?hl=en, 2025.

[35] Microsoft Entra Permissions Management, https://learn.microsoft.com/en-us/entra/permissions-management/, 2025.

[36] E. Marin, D. Perino, and R. Di Pietro, "Serverless computing: a security perspective," Journal of Cloud Computing, vol. 11, no. 1, p. 69, 2022.

[37] Managing Cloud Misconfigurations Risks, https://cloudsecurityalliance.org/blog/2023/08/14/managing-cloud-misconfigurations-risks, 2025.

[38] Simplifying Cloud IAM Policy Management with AI Automation, https://orca.security/resources/blog/multi-cloud-support-iam-policy-optimizer/, 2025.

[39] CodeQL, https://codeql.github.com/, 2025.

[40] N. Eskandani and G. Salvaneschi, "The wonderless dataset for serverless computing," in 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). IEEE, 2021, pp. 565–569.

[41] X. Zhang, Z. Lin, X. Hu, J. Wang, W. Lu, and D.-Y. Zhou, "Secon: Maintaining semantic consistency in data augmentation for code search," ACM Transactions on Information Systems, vol. 43, no. 2, pp. 1–26, 2025.

[42] AWS Identity and Access Management(IAM), https://aws.amazon.com/iam/?nc1=h_ls, 2025.

[43] Google Cloud Identity and Access Management (IAM), https://cloud.google.com/iam/docs/overview, 2025.

[44] Microsoft Azure role-based access control(RBAC), https://learn.microsoft.com/en-us/azure/role-based-access-control/overview, 2025.

[45] deepseek-ai/deepseek-coder-6.7b-instruct, https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct, 2025.

[46] Metric:bleu, https://huggingface.co/spaces/evaluate-metric/bleu, 2025.

[47] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization, 2005, pp. 65–72.

[48] Metric:rouge, https://huggingface.co/spaces/evaluate-metric/rouge, 2025.

[49] M. Popović, "chrf++: words helping character n-grams," in Proceedings of the second conference on machine translation, 2017, pp. 612–618.

[50] J. Wen, Z. Chen, Z. Zhu, F. Sarro, Y. Liu, H. Ping, and S. Wang, "Llm-based misconfiguration detection for aws serverless computing," ACM Transactions on Software Engineering and Methodology, 2025.

[51] S. Arun, M. Tedla, and K. Vaidhyanathan, "Llms for generation of architectural components: An exploratory empirical study in the serverless world," in 2025 IEEE 22nd International Conference on Software Architecture (ICSA). IEEE, 2025, pp. 25–36.

[52] ALPS https://github.com/cclab-inu/ALPS, 2025.

[53] ALPS Dataset https://huggingface.co/datasets/cclabinu/alps, 2025.