Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

AUSPO: Research on Non-Intrusive Microservice Policy Generation Technology

Zhangwen Li
Information Engineering University
Xuehui Du
dxh37139@163.com

Information Engineering University
Aodi Liu
Information Engineering University
Chunhui Fu
Information Engineering University

Research Article

Keywords: Microservices, Access Control Policy, eBPF, DBSCAN clustering

Posted Date: May 11th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9551302/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

AUSPO: Research on Non-Intrusive Microservice Policy Generation Technology

Zhangwen Li[1], Xuehui Du[1*], Aodi Liu[1], Chunhui Fu[1]

[1]Henan Province Key Laboratory of Information Security, Information Engineering University, Zhengzhou, 450000, China.

*Corresponding author(s). E-mail(s): dxh37139@163.com;

Abstract

Microservices have achieved broad industrial development because they offer advantages in maintaining software systems. To prevent attackers from reaching resources hosted by other microservices through lateral movement, authorization policies are required to restrict unauthorized access across services. Existing policy generation approaches depend on manual policy configuration, document analysis, historical log analysis, and static code analysis. However, these approaches impose substantial policy maintenance costs, require the availability of supporting documents and logs, and may cause path explosion problems. To address these challenges, this study proposed Adaptive Unified Security Policy Orchestration (AUSPO). It consisted of three core technologies: (1) A microservice communication data collection method built on eBPF-based monitoring(MseBPF) to capture dependency data among microservices to infer inter-service invocation logic; (2) A policy generation algorithm based on DBSCAN clustering(DBPO), which can translate the conversion of communication data into actual policies and generate access control policies between services (3) An cremental policy update mechanism based on the Pod lifecycle (PodUp), which performs policy analysis exclusively on Pods involved in microservice iteration, providing an efficient approach for subsequent policy updates. We deploy AUSPO in the publicly available Bookinfo and Sockshop microservice environments. Experimental results show that the proposed method can generate fine-grained access control rules to block all unauthorized malicious requests. Additionally, it introduces an average processing latency overhead of no more than 3% for microservices, and completes policy updates with an analysis time of no more than 29% of that consumed in initial policy generation.

Keywords: Microservices, Access Control Policy, eBPF, DBSCAN clustering

1 Introduction

With rapid informatization, software systems are evolving toward increasingly powerful functionality, while the number of application programs is expanding exponentially, together imposing higher demands on development and maintenance [1]. Owing to its advantages in maintenance convenience and scalability, microservice architecture has been widely adopted. Traditional monolithic architecture cannot meet complex and dynamic business requirements [2]. To enable the flexible development and maintenance of complex software systems, microservice architecture has been introduced into software design and gradually developed into an emerging application design paradigm. By decomposing a complex software system into multiple independent and autonomous services, microservice architecture can help debug and develop individual microservices during system maintenance and upgrading, thereby providing notable flexibility and maintainability.

However, the system is composed of numerous highly coupled microservices that invoke one another, and these complex network communication relationships create opportunities for attackers to perform lateral movements [3]. This means not only that the vulnerabilities in a single service can be exploited and rapidly amplified, but also that the application software built on microservice architecture is exposed to a broader potential attack surface than traditional architecture, thereby increasing the complexity of security protection. When attackers break through the API gateway and enter the interior of the microservice system, they may launch attacks against other services with the help of compromised microservices and obtain access to important and sensitive core resources.

To mitigate security risks among microservices, the application of fine-grained access control policies has become a core mechanism for safeguarding the software security of microservice application architectures. As a critical factor in microservice operation, access control rules enhance the security of microservice software by defining inter-service access relationships and limiting legitimate access to sensitive resources to a restricted set of services via predefined URL paths. However, existing access control policies between microservices face several challenges and limitations.

1. The analysis of microservice dependency relationships relies heavily on source code [4]. Static analysis generally requires a complete source code and builds context. In real-world environments involving multi-language technology stacks, dynamic configurations, and partially closed-source components, cross-service invocations or dataflow semantics are prone to omission, which can cause incomplete or inaccurate policy generation. To support rapid scalability, microservice systems are commonly deployed in containers that hold only runtime files associated with the source code, making the original source code difficult to obtain.
2. Numerous policies are redundant [5]. Existing microservice policy enforcement is typically implemented using Linux iptables [6] and microservice infrastructure, including Kubernetes [7] and Istio [8]. When an access request is received, these policy enforcement methods sequentially match the incoming request with existing policies to determine whether the request is legitimate or malicious. In

large-scale microservice systems, linear matching caused by a large number of redundant policies introduces substantial performance overhead and reduces the efficiency of complex inter-service communication.

3. Policy maintenance is costly with frequent updates. In large-scale microservice systems, service versions are updated rapidly, and service dependency relationships are numerous and complex [9]. Under rapid version iteration and topological changes in microservices, statically or manually configured rules are not only extremely time-consuming to maintain [10], but also frequently lag behind the evolution of interfaces and behaviors, thereby increasing the likelihood of policy drift and misconfiguration.

To address the above problems, this study proposed Adaptive Unified Security Policy Orchestration (AUSPO). This system did not rely on source code to analyze microservice dependency relationships. Instead, it adopted a method that mounted Extended Berkeley Packet Filter (eBPF) on the virtual network cards of kernel containers, thereby collecting sufficient inter-microservice access request data in the kernel to support access control policy generation. It employed the policy generation algorithm based on DBSCAN clustering(DBPO) to generate accurate access control policies and finally updated policies through an cremental policy update mechanism based on the Pod lifecycle(PodUp) to address the high maintenance cost caused by frequent subsequent microservice updates. A prototype of AUSPO was implemented on Kubernetes, and its efficiency, security, and resource consumption were evaluated using general open-source microservice datasets.

2 Related Concepts

2.1 Threat Model

In Kubernetes-based microservice environments, lateral movement attacks are one of the most severe security threats [11]. This attack pattern describes a scenario in which an attacker fails to immediately steal or destroy data after successfully compromising and controlling a service instance in the cluster. Conversely, it uses the compromised instance as an initial foothold to probe and infiltrate other services within the same network plane by exploiting the generally high level of trust in the internal cluster networks. Because invocation relationships among microservices are dense and complex, and because default Kubernetes network policies usually permit free communication between containers, attackers can perform lateral movements with relative ease. Through persistent attack techniques such as scanning internal ports, stealing service credentials, exploiting unauthorized API interfaces [12], and launching man-in-the-middle attacks, they may eavesdrop on or even tamper with inter-service communication data. These attacks are highly covert and can exert extensive influence. A security vulnerability in an edge service may ultimately compromise core business data or even the entire infrastructure.Therefore, effective defense against lateral movement attacks requires the implementation of the zero-trust security principle,

enforcement of strict network policies for micro-segmentation, and integration of continuous behavior monitoring and threat detection mechanisms, thereby establishing a defense-in-depth system within the dynamic microservice network.

Using the procurement system shown in Figure 1 as an example, the system consists of four core microservices: Login Service, Asset Information Service, Purchase Order Service, and Vendor Management Service. Under normal business operations, once authentication is completed through the Login Service, legitimate users can obtain vendor information through the Purchase Order Service to create orders or directly query asset information. Notably, the Purchase Order Service is authorized to invoke the interfaces of the Vendor Management Service to obtain vendor details, whereas the Asset Information Service should not directly access vendor data.

Assuming that an attacker compromises the Asset Information Service through a vulnerability and then attempts to access other services by exploiting the default internal trust relationship among microservices. In the absence of fine-grained inter-service access control in the system, the attacker can easily initiate requests from the Asset Information Service to the Purchase Order Service and utilize the permissions already possessed by that service to indirectly invoke the originally protected APIs of the Vendor Management Service, thereby stealing sensitive business information (e.g., vendor contracts, bank account details, and pricing strategies). However, these threats can be mitigated by implementing strict access control policies. For instance, a policy that permits only the Purchase Order Service can be adopted to invoke the Vendor Management Service.

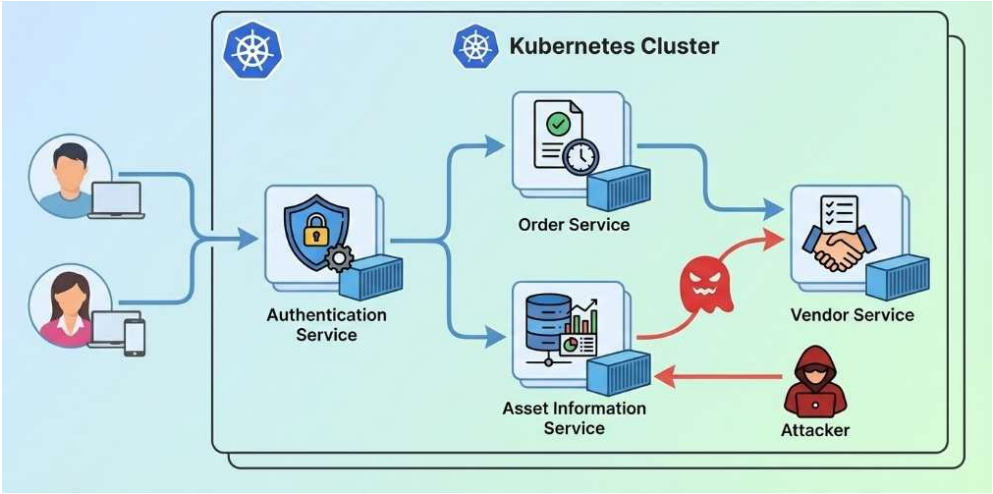


Fig. 1 Example of Inter-Microservice Access Control

To defend effectively against such lateral movement attacks and preserve the security of microservice architecture, exploring methods capable of automatically generating and dynamically updating fine-grained inter-service access control policies is

crucial for ensuring the sustained application of the principle of least privilege in complex microservice environments.

2.2 Microservices

Microservice architecture is an outstanding software design paradigm whose core idea is to decompose a single large monolithic application into a collection of small, loosely coupled, and highly autonomous services [13]. Each service is organized around a specific business domain or functional boundary and maintains its own data storage, business logic, and runtime processes. Through well-defined lightweight communication protocols, such as HTTP-based RESTful APIs [14], high-performance gRPC, and asynchronous message queues, these services collaborate to form a complete application. Compared with traditional monolithic architecture, microservices promote independent development, testing, and deployment across teams through clear service boundaries, thereby substantially improving organizational agility and flexibility in technology selection.

Microservices are typically managed using Kubernetes [15], commonly abbreviated as K8s. It has become the de facto standard for container orchestration in the cloud-native era. It provides a powerful and abstract platform for automated management, deployment, and scaling of microservices. K8s manages containerized applications rather than individual containers directly. On the other hand, it organizes containers through Pods. In a microservice architecture, each microservice instance may correspond to one Pod. Although a Pod may contain multiple containers, it generally runs only one primary business container.

Microservice instances run inside Pods and are exposed to other microservices or external users through K8s Services.

2.3 eBPF

Extended Berkeley Packet Filter (eBPF) is a kernel technology [16] that allows user-written sandboxed programs to be dynamically injected into the Linux kernel for secure and efficient packet processing at the network driver layer. Within the eBPF framework, the system predefines multiple types of hook points for different monitoring and processing scenarios. Kernel probes (kprobes) are mainly applied for tracing and sampling during the execution of specified kernel functions, whereas the corresponding user-space probes (uprobes) are designed to hook functions into user-mode applications. For network packet processing, the traffic controller (tc) hook point enables policy control and traffic shaping for ingress and egress traffic at the network interface controller (NIC), whereas the eXpress Data Path (XDP) hook point is intended for ultra-high-speed filtering, forwarding, or modification of ingress traffic at the driver layer [17]. Each hook point corresponds to a distinct application scenario and performance characteristic. After loading an eBPF program into the host kernel, where the microservices reside, the predefined hook points are activated to record communication data related to microservice dependency relationships.

2.4 DBSCAN Clustering Algorithm

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a classic density-based clustering algorithm whose central idea is that clusters are formed by high-density regions connected through density-reachable relationships, whereas noise points in low-density regions can be effectively identified and separated [18]. The algorithm defines core points through two parameters, namely the neighborhood radius ϵ and the minimum number of points, MinPts, and discovers clusters of arbitrary shapes through the density connectivity of these core points. By applying this clustering algorithm, similar policies can be aggregated, thereby reducing the number of policies and further improving policy enforcement efficiency.

3 Current Research Status

Policy generation can be divided into two main types, namely manual configuration and automatic generation. For microservice and service mesh scenarios, Chandramouli proposed a manual policy configuration method [19]. This approach requires manual assignment of core attributes such as labels, namespaces, environments and identities. In the meantime, researchers need to manually compile RBAC rules, ABAC rules and YAML policies for Sidecars and Envoys. However, this method exhibits limited scalability and incurs high maintenance expenses. As the number of services, versions, and environments continues to increase, the associated rules and attributes expand rapidly, making cross-team rule maintenance highly error-prone and rendering change and rollback operations costly. For IoT scenarios, Qiu proposed an access control model design method [20] that requires the manual definition of access policies, roles, and attributes for massive numbers of devices, along with the manual distribution and synchronization of policies at gateways or in the cloud. However, when device locations, network states, trust levels, and operating modes change frequently, manual updates may lag behind such changes, easily resulting in over-authorization or false rejection and failing to satisfy the requirements of real-time linkage and zero trust. These studies indicate that in dynamic and large-scale microservice environments, manual policy formulation is inefficient and error-prone, and complete dependence on manual policy formulation and management can no longer accommodate the rapid iteration and complex dependency relationships of microservices.

A large body of research has also focused on automatic policy generation, and these studies can generally be grouped into three main categories, including document-based inference methods, policy mining methods driven by historical communication logs, and static analysis techniques based on source code functions related to microservices. The first category consists of document-based methods [21, 22], which primarily rely on Natural Language Processing (NLP) techniques to infer application behavior patterns from textual materials such as system documents and API documentation. Although access rules within a system can be inferred from such documents, the accuracy of policy generation is strongly constrained by the completeness and timeliness of document updates. In practical projects, documentation often lags behind code

implementation or contains inaccurate descriptions. Meanwhile, the inherent ambiguity of natural language may cause the model to misinterpret document intent, thereby producing erroneous policy inferences when complex security constraints are involved.

The second category includes policy mining methods based on historical communication logs, which derive microservice dependency relationships from system logs or other files reflecting historical operations to generate security policies. Log2policy [23] adopts a machine-learning-based attribute mining method to extract the invocation logic between microservices from access logs and generate authorization policies. It achieves the transformation from access logs to inter-microservice invocation relationships through a log-based topology graph generation mechanism and introduces a traffic management mechanism for policy updates, thereby reducing the overhead of subsequent policy upgrade. P-DIFF [24] infers access control policies through continuous monitoring of access logs. To address the challenge of policy evolution, a novel time-varying decision tree is designed to represent changes in access control policies effectively, and it is combined with a new learning algorithm to infer the tree from access logs. Karimi et al. [25] proposed a method for automatically learning ABAC policy rules from system access logs to simplify the policy development process. Their method adopts an unsupervised learning algorithm to detect patterns in access logs and extract Attribute-Based Access Control (ABAC) authorization rules from those patterns. In addition, two policy improvement algorithms, namely rule pruning and policy optimization, are introduced to generate higher-quality mined policies. However, if the historical data contain access records involving security violations or configuration errors, the policies generated may inherit and solidify those errors, thereby leading to persistent security vulnerabilities. Moreover, for newly deployed systems or services, insufficient historical data make effective policy generation difficult and require a long period of data accumulation.

The third category comprises code-based methods [26], which use static analysis techniques to extract requests and their attributes from the source code of microservice applications. Li [27] developed a static-analysis-based request extraction mechanism that combined program slicing with semantic analysis to derive detailed inter-service invocation logic. Static code analysis is employed to examine containerized applications and all their dependencies and to identify the system call resources required for their correct operation. Although source code analysis can reveal the logic connecting distributed software or software processes, microservice source code is often excessively complex for developers, and learning how to invoke microservice code entails substantial time costs and may also cause the path explosion problem. Zanatta et al. [28] proposed a policy generation method for the Robot Operating System 2 (ROS2), namely LiSA4ROS2, ROS2 Security Policy Automated Extraction, which infers the complete computation graph of ROS2 applications, that is, all nodes and their communication relationships, through source-code analysis. This method uses static-analysis-based abstract interpretation theory to approximately deduce the complete communication behavior of nodes, thereby ensuring the completeness of the generated permission files while making them as precise as possible using algorithms. Although this method was designed for ROS2, it provides useful insights for generating complete policy files. However, the third category of methods may fail to

obtain all requested attributes and impose high requirements on source code quality. In large-scale microservice systems, full-code static analysis requires substantial computational resources, and the analysis time may increase exponentially with the code scale. In addition, code analysis often ignores external configuration files, database constraints, and the permission logic of third-party services, thereby resulting in incomplete generated policies.

In summary, current methods still exhibit limitations. Document-based methods are suitable for the system design stage but are constrained by document quality. Log-based policy mining methods can reflect actual access patterns, but they suffer from cold-start problems and the risk of historical error propagation. Code-based methods provide in-depth program analysis; however, they require developers to understand complex source codes, which consumes considerable time, and source-code analysis may also trigger the path explosion problem.

AUSPO employs eBPF technology to derive microservice dependency relationships without relying on logs or source code, thereby avoiding the cold-start problem, risk of historical error propagation, and path explosion problem. It further applies the DBSCAN clustering algorithm to merge similar policies, thereby reducing the number of policies. During policy enforcement, access requests are matched against policies in a linear manner; therefore, reducing the number of policies can improve access control efficiency. In addition, the graph-based policy update mechanism enables policy updates to be completed at a minimal cost during subsequent microservice updates and iterations.

4 Design of AUSPO Core Technologies

4.1 eBPF-based Kernel Data Collection Method MseBPF

eBPF-based data collection serves as the core prerequisite for security policy generation. In this phase, XDP-powered eBPF programs are adopted to monitor the Virtual Network Interface Cards (vNICs) of Pods, achieving non-intrusive capture of communication data among microservices. In Kubernetes-deployed microservices, each version of a microservice is typically hosted in an independent Pod, and every Pod is attached to a dedicated vNIC. When a microservice communicates with others, the vNIC receives and forwards communication traffic. Thus, attaching eBPF programs to vNICs enables the acquisition of massive microservice communication data.

The architecture of MseBPF is illustrated in Figure 1. First, the developed eBPF probe programs are loaded into the Linux kernel and bound in batches to all screened vNICs. The probe parses data packets passing through the network cards layer by layer in the kernel space, and accurately collects six-tuple data (src_ip, src_port, dst_ip, dst_port, h_method, h_url), representing the source Pod IP address, source Pod port, destination Pod IP address, destination Pod port, HTTP method, and URL path, respectively. To avoid introducing extra performance overhead during collection, the kernel-mode Performance Buffer (Perf Buffer) is utilized to implement asynchronous data transmission to the user space. This transmission mechanism effectively avoids frequent mode switching between the kernel space and user space, guaranteeing high-efficiency data transmission and low overhead in the collection process. During data

collection, the probe only focuses on traffic capture without performing any real-time semantic mapping operations, further reducing the consumption of system resources.

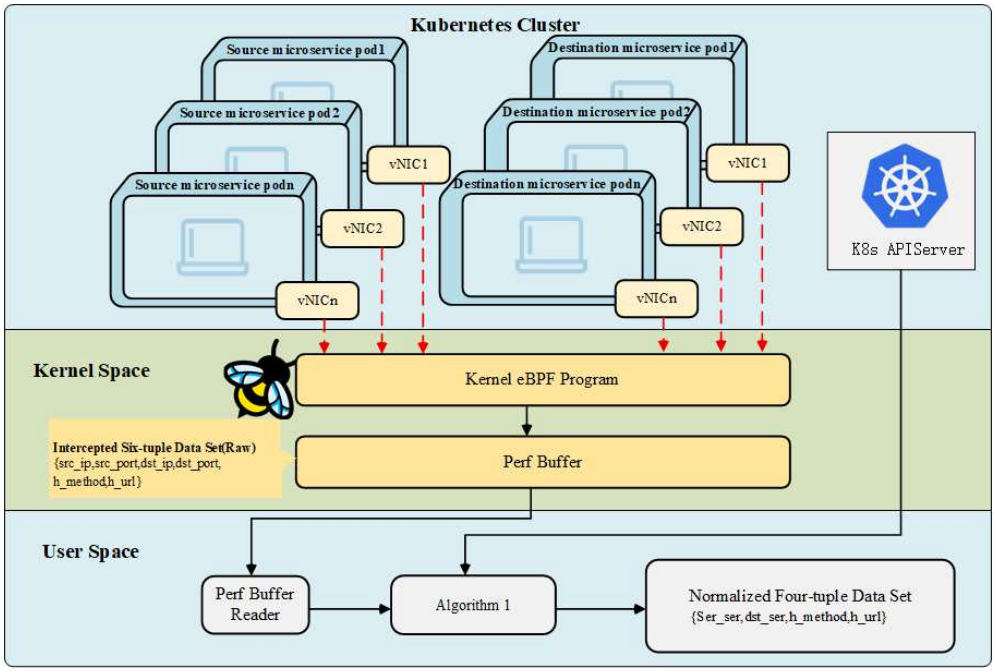


Fig. 2 eBPF Data Collection Process

In the user space, the application receives data transmitted from the kernel via the Perf Buffer. However, duplicate six-tuple records commonly exist in the collected data, so unique identification and deduplication must be performed on the traffic to eliminate redundant data and improve the efficiency and accuracy of subsequent data processing. Meanwhile, in order to facilitate policy generation through subsequent clustering, the six-tuple of communication data is converted into a four-tuple (src_ser, dst_ser, h_method, h_url), denoting the source microservice name, destination microservice name, HTTP method, and URL path, respectively. The raw six-tuple data collected by eBPF contains abundant low-level network redundant information irrelevant to security policy generation. For the generation of microservice access control policies, the core requirement is only to clarify which microservice accesses another microservice via which HTTP method and URL path, and the four-tuple can precisely satisfy this core demand. To solve data redundancy and six-tuple-to-four-tuple conversion issues, MseBPF designs Algorithm 1 by combining the flow hashing algorithm with the batch binary search algorithm. With the eBPF-collected raw six-tuple traffic data and K8s API Server interfaces as inputs, the algorithm first initializes a hash table unique_flows for storing unique traffic records, a mapping table ip_service_map between Pod IPs and microservice names, and a sorted IP list sorted_ips. Subsequently, it traverses all raw

traffic records, generates a unique hash key using the six-tuple of each traffic entry, and stores the hash key and complete traffic record in the hash table. The uniqueness of hash table keys automatically overwrites duplicate records, completing traffic deduplication and obtaining a redundant-free six-tuple dataset. Next, it pulls the mapping relationships between IPs and microservice names of all Pods in the cluster in batches through the K8s APIServer, extracts and sorts all IP addresses from the mapping table to construct an ordered IP list. Finally, it traverses the deduplicated traffic records, performs binary search on the source IP and destination IP of each record respectively, quickly locates the index position corresponding to the IP in the ordered IP list, and obtains the source and destination microservice names matched with the IP from the mapping table. The source microservice, destination microservice, HTTP method, and URL path are combined into a standardized four-tuple and stored in the result set, ultimately returning a complete standardized four-tuple dataset.

Algorithm 1 MseBPF Data Processing Algorithm

Require: Raw six-tuple traffic $raw_traffic$, K8s APIServer $k8s_api$
Ensure: Standardized four-tuple dataset $Result$

$unique_flows \Leftarrow$ empty hash table
$ip_service_map \Leftarrow$ empty mapping table
$sorted_ips \Leftarrow$ empty sorted list
for each $flow$ in $raw_traffic$ **do**
    $hash_key \Leftarrow hash(flow.(src_ip, src_port, dst_ip, dst_port, h_method, h_url))$
    $unique_flows[hash_key] \Leftarrow flow$
end for
$ip_service_map \Leftarrow k8s_batch_get_pod_info(k8s_api)$
$sorted_ips \Leftarrow sort(ip_service_map.keys())$
for each $flow$ in $unique_flows$ **do**
    binary_search($sorted_ips$, $flow.src_ip$)
    $src_ser \Leftarrow ip_service_map[flow.src_ip]$
    binary_search($sorted_ips$, $flow.dst_ip$)
    $dst_ser \Leftarrow ip_service_map[flow.dst_ip]$
    $quadruple \Leftarrow (src_ser, dst_ser, flow.h_method, flow.h_url)$
    $Result.append$(quadruple)
end for
return $Result$

By combining flow hashing deduplication and batch binary search, the proposed algorithm achieves two key optimizations. On one hand, it leverages the $O(1)$ lookup characteristic of hash tables to reduce the complexity of deduplication to $O(n)$, thereby efficiently eliminating redundant traffic from repeated requests. This provides high-quality, interference-free input data for subsequent DBPO density clustering and fundamentally avoids the distortion of clustering results caused by redundant data. On the other hand, by integrating batch fetching of Kubernetes metadata with binary search, the algorithm optimizes the query complexity of IP-to-microservice name

mapping from $O(n)$ in traditional traversal to $O(\log n)$, which significantly improves the efficiency of semantic mapping in large-scale cluster scenarios. Meanwhile, the offline batch processing manner avoids the performance overhead incurred by real-time queries to the Kubernetes APIServer, which is highly consistent with the core design philosophy of low-overhead eBPF collection. Furthermore, the transformation from six-tuple to four-tuple effectively filters out low-level network redundant information while completely preserving the business semantics of microservice access. This allows clustering results to be directly mapped into microservice access control policies, forming a complete technical closed-loop from traffic collection to policy generation. The proposed method notably enhances the rigor, innovation, and engineering practicability of this research, offers reliable support for the technical pipeline of this paper, and makes the research outcomes better align with the application requirements of real-world Kubernetes microservice clusters.

4.2 Policy generation algorithm based on DBSCAN clustering DBPO

A large volume of four-tuple data generated from microservice communication is obtained via the MseBPF method. To convert the four-tuple dataset into corresponding security policies, this paper proposes the DBPO algorithm. Converting the four-tuple dataset into policies faces two critical challenges. First, numerous four-tuple entries share the same source and destination microservices but differ in HTTP methods and URLs. Generating an independent policy for each distinct four-tuple will introduce excessive overhead to access control enforcement. Second, indiscriminately fusing all entries into a single policy leads to severe over-generalization and the critical issue of over-authorization. Therefore, this paper adopts the DBPO algorithm, which is built upon the unsupervised learning-based DBSCAN algorithm, for policy generation. The generated policies can be directly transformed into Kubernetes authorization policies, realizing closed-loop automation from real traffic observation to security policy generation. This mechanism significantly enhances the security and operational efficiency of cloud-native environments.

4.2.1 DBPO Data Processing Method

The massive four-tuple access dependency data collected by MseBPF requires preprocessing prior to clustering. In this paper, all four-tuple access request data acquired by MseBPF are classified based on the source service name and destination service name. Subsequently, the DBSCAN algorithm is employed to perform clustering on the HTTP methods and URL paths.

HTTP methods as a feature dimension can be quantified through a fixed-value mapping approach, with each method assigned a specific numerical value (Table 1).

Each URL was abstracted into an 11-dimensional feature vector, and the corresponding dimensional design and quantification rules were defined as follows. The first dimension was obtained by counting the number of the delimiter in the URL, thereby representing the actual depth of the URL path. The URL path was extended to a fixed format containing 10 delimiters . If the original path contains fewer delimiters,

Table 1 HTTP Method Quantification.

Method	Value	Method	Value	Method	Value	Method	Value
GET	0.05	PATCH	0.30	MOVE	0.55	UNLINK	0.80
POST	0.10	CONNECT	0.35	PROPFIND	0.60	LINK	0.85
PUT	0.15	TRACE	0.40	PROPPATCH	0.65	MKCOL	0.90
DELETE	0.20	COPY	0.45	UNLOCK	0.70	OPTIONS	0.95
HEAD	0.25	LOCK	0.50	PURGE	0.75	-	-

the character & can be used for padding. For instance, the URL /api/cart/44553 was padded to /api/cart/44553/&/&/&/&/&/&/&. The 2nd to 11th dimensions were used to quantify the 1st to 10th segments of the padded URL, with each segment separated by /. The quantification rules were defined as follows. The numeric segments were mapped to 0; the padding segments represented by & were mapped to 1; and the alphabetic segments were calculated using Equation (1), where the value is the sum of the products of each letter's position in the alphabet and its weight within the segment.

Using the padded URL /api/cart/44553/&/&/&/&/&/&/& as an example, it was quantified as an 11-dimensional vector(3, 60, 140, 0, 1, 1, 1, 1, 1, 1).

$$\sum_{i=1}^{n} Le^{i}_{\text{positron}} \cdot Le^{i}_{\text{mode}} \tag{1}$$

The quantified HTTP method and the URL feature vector were concatenated to form a 12-dimensional feature vector for each access request. Before introducing this vector into the DBSCAN algorithm, Equation (2) was used for standardization (where u denotes the standard deviation of the dimension, and σ denotes the variance) and Equation (3) for normalization, thereby eliminating the influence of differences in the feature scales.

$$x_{\text{standardization}} = \frac{x - \mu}{\sigma} \tag{2}$$

$$x_{\text{normalization}} = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{3}$$

4.2.2 Clustering-based Policy Generation

By quantifying the features of HTTP methods and URLs, each access request can be converted into a 12-dimensional feature vector, consisting of 1-dimensional quantized value of HTTP methods and 11-dimensional URL feature vectors. The feature vectors of all samples collectively form a feature matrix, which serves as the input for the DBSCAN density clustering algorithm.

DBSCAN is selected as the clustering algorithm owing to its core advantages: it requires no pre-specification of the number of clusters, can automatically identify noise points based on density, and groups samples with similar access characteristics into the same cluster. This is highly consistent with the assumption in microservice privilege

management that service subjects of the same type should possess similar privilege sets. Samples within the same cluster correspond to highly similar access behaviors from the source service to the destination service and can share a unified privilege policy.

The DBSCAN algorithm divides samples into different clusters based on density reachability by calculating the Euclidean distance between samples and generates a cluster center for each cluster. The distance from samples within the same cluster to the cluster center is smaller than the preset threshold, indicating their highly similar access characteristics. Accordingly, the access behaviors of all samples in a cluster can be aggregated into a representative policy: extracting the union of all HTTP methods and the longest common prefix of URL paths within the cluster to generate the least privilege rule covering all access behaviors of the cluster. For instance, if a cluster contains samples such as GET/api/cart/456 and DELETE/api/cart/789, it can be aggregated into the policy GET,DELETE/api/cart/*, compressing the original massive four-tuple policies into a small number of highly generalized rules.

After clustering, samples in the same cluster are regarded as a set of access behaviors with identical privileges, and their feature vectors exhibit high similarity, implying highly consistent access permission requirements for the target service. However, DBSCAN clustering only outputs clusters of samples with similar access behaviors. To transform the cluster structure into available access control policies that comply with the least privilege principle for microservice applications, this study designs a three-level transformation rule to achieve accurate mapping from cluster features to generic policies, while ensuring the fine granularity and non-over-authorization of policies.

The first level is the intra-cluster core feature extraction stage. For each valid cluster, two categories of core features are first extracted to provide fundamental support for subsequent policy generation. The first is the HTTP method set, which takes the union of HTTP methods such as GET and POST of all samples in the cluster to ensure full coverage of all legitimate operation types without omitting normal access behaviors. The second is the URL path feature, used to extract the hierarchical structure and fixed segments of all URLs in the cluster. For URL path feature extraction, all URLs in the cluster are first split into path segments by slashes. Taking GET /api/order/123 and GET /api/order/456 as examples, they are split into path segment sequences of api, order, 123 and api, order, 456 respectively. Then, segments at the same hierarchy are compared layer by layer: if a segment at a certain hierarchy is a fixed string such as api and order, it is retained as the fixed part of the path; if the segment is a variable instance such as 123 and 456 characterized by extremely close vector distances but different values at the hierarchy within the cluster, it is marked as a variable position. Finally, a path template is generated based on the variable marking results, and the path template corresponding to the above example is /api/order/*.

The second level is the generic policy generation and merging stage. Based on the extracted core features, generic policies are generated according to service-oriented behavior types. First, the policy subject is defined as the access relationship from the source microservice to the destination microservice corresponding to all samples in the cluster including the call link of order-service accessing product-service to ensure the service granularity accuracy of the policy. Subsequently, the HTTP method set and the

URL path template are combined to form a basic policy, such as allowing order-service to access the GET /api/order/* path of product-service, completing the definition of operation permissions. For the merging of similar policies, if the same service pair corresponds to multiple clusters belonging to different behavior types including read, write and high-risk operations, cross-cluster policy merging is not performed, and only independent policies of each cluster are retained including POST /api/order/* as a separate policy. This avoids the confusion of operations with different risk levels and strictly follows the least privilege principle.

Finally, policy validity filtering is performed. Two supplementary filtering mechanisms are adopted to avoid over-authorization or invalid policies during policy transformation. The first is variable position verification: a segment is marked as a variable only if the vector distance of segments at the same hierarchy is less than $\epsilon/2$, ensuring that the marked object is a real variable instance rather than different fixed segments, and preventing mis-marking structurally different paths as variables including /api/order/123 and /api/user/456 that will not be merged. The second is empty policy elimination: if a cluster contains only a single sample with no similar paths, a precise matching policy such as GET /123/order/api is generated instead of forced wildcard processing, avoiding over-authorization from the source.

The policy rules generated based on the DBPO algorithm not only ensure the integrity of permission coverage but also avoid over-authorization, complying with the least privilege principle. Meanwhile, the significant reduction in the number of policies drastically decreases the runtime policy matching overhead and improves the efficiency and scalability of permission decision-making in microservice architectures.

4.3 Cremental policy update mechanism based on the Pod lifecycle PodUp

The fine-grained access policies generated via clustering inherently align with the Kubernetes deployment specification, where a single Pod corresponds to an individual microservice instance or version. To address policy synchronization requirements in scenarios such as microservice version iteration and dynamic instance scaling, this study proposes PodUP, a Pod event-driven incremental policy update mechanism. The mechanism achieves efficient policy adjustment through cluster feature reuse and dynamic rule adaptation, eliminating full re-clustering while consistently ensuring access rule accuracy and compliance with the least privilege principle.

4.3.1 Incremental Policy Adaptation for Pod Provisioning Scenarios

When new Pods are provisioned in the cluster due to version iteration or service scaling, the system eliminates the need to re-execute the full clustering process and generates policies incrementally by leveraging core parameters of existing clusters. The system locates the microservice and version information of new Pods by listening to Pod creation events via the Kubernetes API or identifying newly added vNICs monitored by eBPF. Subsequently, it only collects incremental traffic for the collection module mounted to the target Pod, converting the traffic into a 12-dimensional feature

vector: 1 dimension for the HTTP method type, and 11 dimensions for the quantized features of the URL path.

For scenarios where only partial behaviors of the new Pod are consistent with existing Pods, the system adopts a per-flow vector matching approach for split adaptation, rather than making a unified judgment on the overall traffic. For each incremental flow, the Euclidean distance between its feature vector and the center of existing clusters is calculated using the following formula.

$$\text{dist}(\mathbf{x}, \mathbf{c}_j) = \sqrt{\sum_{k=1}^{12}(x_k - c_{jk})^2} \tag{4}$$

where $\mathbf{x}$ denotes the 12-dimensional feature vector of the incremental flow, $\mathbf{c}_j$ denotes the center vector of the j-th existing behavior cluster, and $\text{dist}(\mathbf{x}, \mathbf{c}_j)$ is the Euclidean distance between them. After calculation, flow features with a distance less than the clustering threshold ε are determined to be consistent with existing cluster behaviors, and the corresponding general access rules of the cluster are directly reused. For flows with a distance greater than or equal to ε, their local density is calculated separately. If the local density meets the requirement of the minimum number of neighborhood points MinPts, a new behavior cluster and dedicated policy rules are generated through lightweight density clustering; otherwise, the flow is identified as abnormal noise and discarded directly.

Through this per-flow split matching approach, the new Pod can simultaneously reuse existing mature rules and adapt to new behavior rules, and the integration of the two forms a complete set of access policies. For example, when the order service is upgraded from v1 to v2, the query traffic of the v2 Pod can match the existing read behavior cluster and reuse the corresponding rules, while the newly added batch operation traffic generates independent policies, which are finally merged to form the complete access permissions of the v2 Pod. Once generated, the new policies can be directly synchronized to the permission enforcement layer, enabling rapid permission activation for the new instance.

4.3.2 Policy Cleanup and Rule Stability Guarantee for Pod Termination Scenarios

When a Pod is removed due to service offline or version retirement, the system locates the instance via Pod deletion events or the offline status of the vNIC, and directly cleans up all policy entries associated with the instance without requiring any clustering computation. The system only deletes the instance-level policy index directly bound to the terminated Pod, without modifying core policy contents such as the general URL generalization templates and behavior cluster divisions generated by clustering. Therefore, the core structure and rule logic of the least privilege policy remain unchanged.

Taking the offline of the order service v1 as an example, the system only removes the access rules sourced from the v1 Pod, as well as the invalid rules from other instances pointing to the retired Pod. The general access rules and behavior cluster

divisions formed by clustering remain intact. This achieves immediate revocation of invalid permissions without altering the form and coverage of the minimal privilege policy due to instance termination. The cleaned policy set only retains valid rules for online microservice instances, consistently maintaining the simplicity and consistency of the policy system.

5 Experiments

5.1 Experimental Environment

Our method was implemented in a local Kubernetes environment, and its effectiveness and performance were evaluated using two widely adopted open-source Kubernetes benchmark microservice applications: Bookinfo [29] and Sock Shop [30].

Bookinfo is a microservice application that simulates an online bookstore catalog and ordering system. Sock Shop is an e-commerce microservice application with complex internal logic in which a single request can trigger multiple nested invocations, thereby simulating the user interface of an online sock store.

All microservices were deployed and managed in Containerd (v1.7.28) containers and orchestrated by Kubernetes (v1.30.1) across three local Ubuntu virtual machines. The network communication among cluster nodes was implemented through Calico (v3.28.5). These virtual machines were hosted on a physical server equipped with a 12th Gen Intel(R) Core(TM) i9-12900H 2.50 GHz processor, 8 physical cores, and 64 GB of memory, and each virtual machine was allocated 4 vCPUs and 16 GB of memory.

We conduct a comparative evaluation against two representative approaches, AUTOARMOR [28] and MicroGuard [5]. AUTOARMOR generates security policies through static code analysis and privilege graph management. MicroGuard represents a non-intrusive inter-service access control system that automatically synthesizes access control policies using a Trie structure and the BERT model.

In this section, we evaluate the effectiveness and efficiency of AUSPO. Specifically, our evaluation aims to address the following three research questions. First, how is the quality of access control rules generated from scratch. Second, what is the performance overhead incurred during rule generation. Third, can the proposed policy update mechanism improve update efficiency when microservices are iterated and updated after the initial rules are generated.

5.2 Policy Effectiveness Evaluation

To verify whether the proposed AUSPO approach can accurately derive all dependency relationships among microservices and generate correct access control policies, we conduct security experiments to evaluate the capability of generated policies against lateral attacks. We construct two categories of malicious requests to simulate typical lateral threats in microservice systems, namely unauthorized access to external microservices A1 and unauthorized access to restricted resources A2.

Attack A1 refers to unauthorized access to external microservices, in which the target microservice receives access requests that violate the rules defined in the generated policy. The attacker attempts to move laterally from an accessible microservice to other microservices beyond its assigned privileges. In this study, we launch A1 attacks by generating additional non-existent microservice dependencies through scripts and sending corresponding requests. Attack A2 refers to unauthorized access to restricted resources, in which the attacker attempts to access extra ports and URL paths beyond the scope of access control policies on a legally accessible microservice, so as to invade other sensitive resources of the target service. We perform A2 attacks by generating requests with legitimate service access directions but illegal URL paths and HTTP methods that are not covered by the access policies.

We simulate a total of 10,000 access requests against the two benchmark microservice applications using automated scripts. We adopt a ratio of A1 to A2 attacks of 2 to 8 in the Bookinfo scenario and 3 to 7 in the Sock Shop scenario. The results of the two simulated attack types are summarized in Table 2. Experimental results show that all unauthorized access attempts are successfully blocked, indicating that the policies generated by AUSPO can effectively prevent all unauthorized access behaviors. It can be concluded that the policies produced by our method sufficiently protect critical resources in microservice systems and further validate that AUSPO can precisely capture inter-service dependencies and generate reliable access control policies.

Table 2 Results of Two Types of Simulated Attacks.

Application		Bookinfo		Sock Shop	
		Success	failure	Success	failure
A1	Requests	2000 (20%)		3000 (30%)	
	Blocked requests	2000	0	3000	0
A2	Requests	8000 (80%)		7000 (70%)	
	Blocked requests	8000	0	7000	0
Total	Requests	10000 (100%)		10000 (100%)	
	Blocked requests	10000	0	10000	0

5.3 Performance evaluation

5.3.1 Analysis of policy implementation time cost

In this study, we compare the policy analysis time of AUSPO with two representative baseline approaches, namely AUTOARMOR and MicroGuard. As presented in Table 3, AUTOARMOR consumes 174 seconds to generate access policies, while MicroGuard takes 25 seconds. In contrast, our proposed method only costs 20.04 seconds. The results demonstrate that our approach outperforms existing methods, reducing the analysis time by at least 24%. Such performance improvement benefits from the fact that our method relies neither on source code static analysis nor on BERT-based policy prediction. It further verifies that the DBPO algorithm adopted in AUSPO effectively saves considerable time overhead for policy generation.

Table 3 Time required for policy analysis

Application	AUTOARMOR	MicroGuard	AUSPO		
			DBPO	Policy Output	Total
Bookinfo	78s	5.23s	3s	2s	6s
Sock Shop	358s	146.25s	100s	3s	103s

5.3.2 Comparison of Data Collection Resource Consumption

Throughout the experiment, the CPU utilization of the two worker nodes running AUSPO was continuously monitored by collecting CPU metrics every 5 min and calculating the average utilization rate. Figure 3 presents the average CPU utilization of the two microservice applications with and without AUSPO. Taking Bookinfo as an example, when AUSPO was not enabled, the microservice containers on Worker Node 1 consumed 3.5% of CPU resources, whereas those on Worker Node 2 consumed 1.9%. After AUSPO was enabled, the CPU utilization on the two nodes increased by 1.0% and 0.6%, respectively, and a similar pattern was observed in Sock Shop.

This slight CPU overhead arises from the additional processing introduced by the data collection. Specifically, the system should filter all network traffic to extract inter-microservice communication features, which could introduce a certain amount of kernel-mode computational cost, mainly associated with the network traffic filtering and data collection logic executed by the eBPF probes in the AUSPO framework. Nevertheless, the experimental results demonstrate that the average CPU overhead introduced by the AUSPO framework is merely 0.873%, which is significantly lower than the average overhead of 5% to 10% incurred by Istio, the conventional Sidecar proxy solution. This overhead falls within the acceptable range for production environments and imposes no significant impact on the core business computing of microservices.

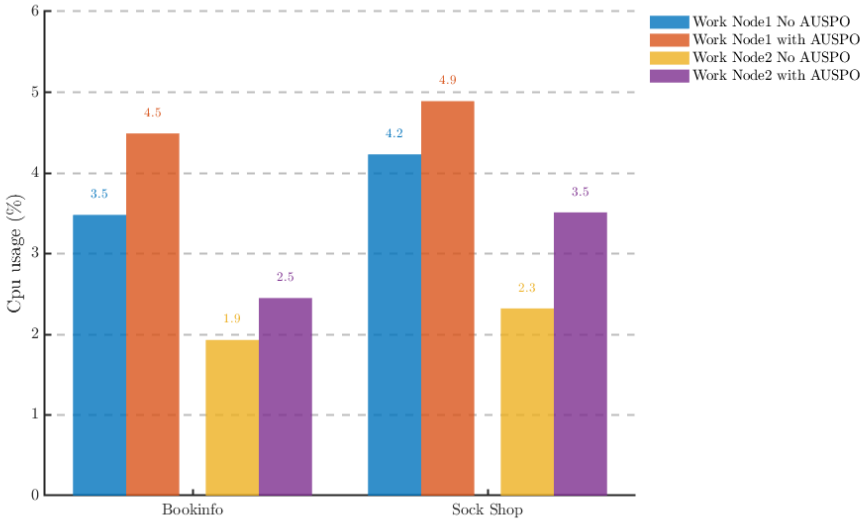


Fig. 3 Comparison of CPU Utilization.

To evaluate the impact of AUSPO on inter-service communication performance, the communication latency between core microservices in both Bookinfo and Sock Shop was measured. The experiment adopted the HTTP request?response model, with 1000 repeated invocations performed for each service pair, and the average latency was calculated to fully reflect the framework?s impact on communication performance.

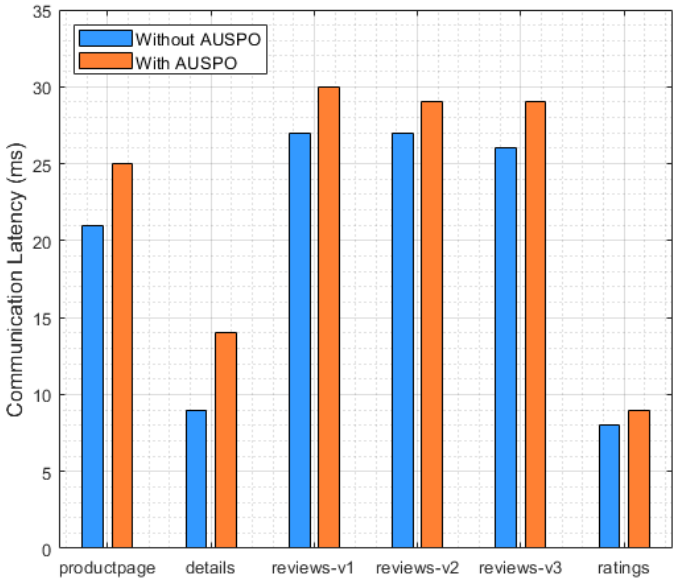


Fig. 4 Response Latency Comparison for Bookinfo with and without AUSPO.

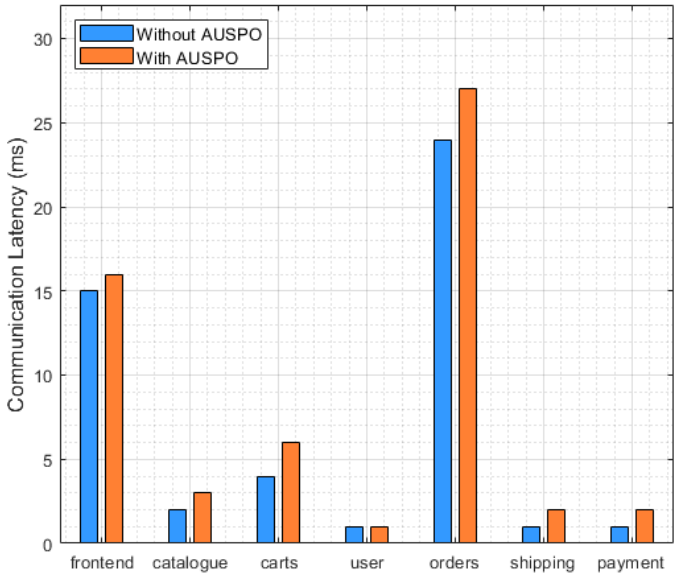


Fig. 5 Response Latency Comparison for Sock Shop with and without AUSPO.

The experimental results are shown in Figures 4 and 5. Overall, after AUSPO was deployed, the average increase in inter-microservice communication latency did not exceed 3%. This slight increase in latency was mainly attributed to the minimal overhead introduced by the packet-processing logic of the eBPF probe and user-mode event handling. When packets were parsed and filtered in kernel mode, the eBPF program introduced a sub-millisecond processing latency. However, because the eBPF program is executed efficiently in kernel mode, the framework avoids the substantial overhead caused by repeated packet copying and forwarding between user and kernel modes in traditional Sidecar solutions, thereby maintaining the overall latency increase within an acceptable range and producing no significant effect on business response time.

5.4 Policy update efficiency

We evaluate the efficiency of AUSPO when upgrading access control rules. We design two scenarios for the Bookinfo microservice application, including adding one microservice with two new versions and removing an existing microservice. These operations correspond to adding one Pod, deploying two new Pods, and deleting one Pod in Kubernetes. For the Sock Shop microservice application, we also design two scenarios: introducing three new microservice versions and removing one microservice, which map to adding three Pods and deleting one Pod in Kubernetes respectively. The baseline approach involves the normal deployment of new versions or services,

followed by the collection of communication data from all Pods and the generation of access control rules via DBPO.

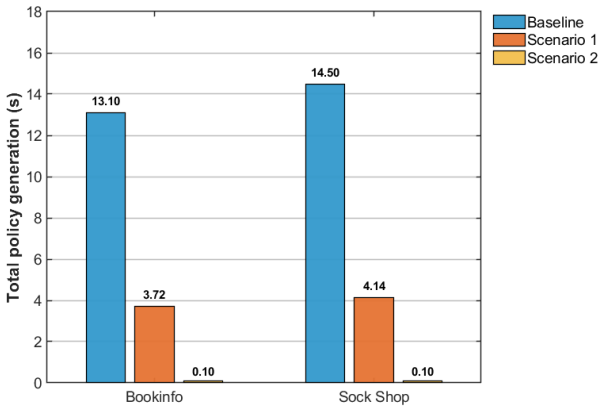


Fig. 6 Policy update comparison

Experimental results are illustrated in Figure 6. Both the baseline method and the policy update using PodUP can generate sound and fine-grained access control rules for the updated microservices. However, PodUP consumes at most 29% of the time required by the baseline. Since PodUP avoids collecting communication data from all Pods and generating irrelevant policies for unchanged microservices during policy updates, its upgrade efficiency is significantly improved. Higher upgrade efficiency reduces the time window in which microservice applications are left unprotected by the access control mechanism.

6 Conclusion

This study proposes an eBPF-based approach for generating fine-grained access control policies for microservices. By deploying lightweight eBPF probes in the Linux kernel space, the approach captures real-time interaction behaviors among microservices in a non-intrusive manner without relying on service source code or static logs, and accurately constructs a service dependency dataset. On this basis, the DBSCAN density clustering algorithm is adopted to quantify and cluster access features such as HTTP methods and URL paths, generating streamlined policies conforming to the principle of least privilege. Moreover, a Pod lifecycle-oriented incremental policy update mechanism is leveraged to provide efficient support for the dynamic update and upgrade of policies.

To verify the feasibility and performance of the proposed approach, a prototype system named AUSPO is implemented on the Kubernetes platform, and comprehensive evaluation experiments are conducted on two popular open-source microservice applications, namely Bookinfo and Sock Shop. Experimental results demonstrate that

AUSPO can rapidly generate correct and secure fine-grained network policies, while the introduced CPU overhead and communication latency overhead are both within the acceptable range for production environments. The proposed approach effectively addresses the core challenges of policy generation and management in microservice architectures.

It should be noted that the current approach only supports the monitoring of HTTP plaintext traffic. For encrypted communication scenarios such as HTTPS, it cannot directly parse the HTTP methods and URL information from the traffic, which is a key limitation of this work. In future research, we will further explore eBPF-based technologies for encrypted traffic parsing and feature extraction to improve the applicability and robustness of the proposed approach, enabling its adaptation to more complex and secure production-level microservice environments.

References

[1] Balalaie, A., Heydarnoori, A., Jamshidi, P.: Migrating to cloud-native architectures using microservices: an experience report. In: European Conference on Service-oriented and Cloud Computing, pp. 201–215 (2015). Springer

[2] Haugeland, S.G., Nguyen, P.H., Song, H., Chauvel, F.: Migrating monoliths to microservices-based customizable multi-tenant cloud-native apps. In: 2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 170–177 (2021). IEEE

[3] Theodoropoulos, T., Rosa, L., Benzaid, C., Gray, P., Marin, E., Makris, A., Cordeiro, L., Diego, F., Sorokin, P., Girolamo, M.D., *et al.*: Security in cloud-native services: A survey. Journal of Cybersecurity and Privacy **3**(4), 758–793 (2023)

[4] Soares, E., Paixao, M., Araújo, A.A.: Microsensor: Towards an extensible tool for the static analysis of microservices systems in continuous integration. In: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, pp. 673–674 (2024)

[5] Luo, H., Yang, W., Chen, P.: Microguard: Non-intrusive dynamic analysis for inter-service access control of microservices. In: Proceedings of the 16th International Conference on Internetware, pp. 245–257 (2025)

[6] Wang, B., Lu, K., Chang, P.: Design and implementation of linux firewall based on the frame of netfilter/iptable. In: 2016 11th International Conference on Computer Science & Education (ICCSE), pp. 949–953 (2016). IEEE

[7] Orchestration—Kubernetes, P.-G.C.: Production-Grade Container Orchestration—Kubernetes (2020)

[8] Authors, I.: Istio-Connect, secure, control, and observe services. nov (2020)

[9] Hillpot, J.: Microservices Examples: Amazon, Netflix, Uber, and Etsy (2023)

[10] Akon, M., Yang, T., Dong, Y., Hussain, S.R.: Formal analysis of access control mechanism of 5g core network. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, pp. 666–680 (2023)

[11] Jarkas, O., Ko, R., Dong, N., Mahmud, R.: A container security survey: Exploits, attacks, and defenses. ACM Computing Surveys **57**(7), 1–36 (2025)

[12] Al Mashta, L.: Containers: Security challenges and mitigation strategies: A systematic literature review (2024)

[13] Alshuqayran, N., Ali, N., Evans, R.: A systematic mapping study in microservice architecture. In: 2016 IEEE 9th International Conference on Service-oriented Computing and Applications (SOCA), pp. 44–51 (2016). IEEE

[14] ESEOGHENE, K., PRINCESS, E.O., OLAKOJO, Y.O., MARK, O.N.: A cybersecurity framework for fraud detection in financial systems using ai and microservices. GULF JOURNAL OF ADVANCE BUSINESS RESEARCH **3**(2), 410–424 (2025)

[15] Pai, P., Kumar, C.: Building cloud native application—analysis for multicomponent application deployment. In: 2021 International Conference on Computer Communication and Informatics (ICCCI), pp. 1–6 (2021). IEEE

[16] Vieira, M.A., Castanho, M.S., Pacífico, R.D., Santos, E.R., Júnior, E.P.C., Vieira, L.F.: Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. ACM Computing Surveys (CSUR) **53**(1), 1–36 (2020)

[17] Vieira, M.A., Castanho, M.S., Pacífico, R.D., Santos, E.R., Júnior, E.P.C., Vieira, L.F.: Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. ACM Computing Surveys (CSUR) **53**(1), 1–36 (2020)

[18] Singh, H.V., Girdhar, A., Dahiya, S.: A literature survey based on dbscan algorithms. In: 2022 6th International Conference on Intelligent Computing and Control Systems (ICICCS), pp. 751–758 (2022). IEEE

[19] Chandramouli, R., Butcher, Z., Chetal, A., et al.: Attribute-based access control for microservices-based applications using a service mesh. NIST Special Publication **800**(S 41) (2021)

[20] Qiu, J., Tian, Z., Du, C., Zuo, Q., Su, S., Fang, B.: A survey on access control in the age of internet of things. IEEE Internet of Things Journal **7**(6), 4682–4696 (2020)

[21] Abdelgawad, M., Ray, I., Alqurashi, S., Venkatesha, V., Shirazi, H.: Synthesizing

and analyzing attribute-based access control model generated from natural language policy statements. In: Proceedings of the 28th ACM Symposium on Access Control Models and Technologies, pp. 91–98 (2023)

[22] Yang, M., Atluri, V., Sural, S., Kundu, A.: Extraction of machine enforceable abac policies from natural language text using llm knowledge distillation. In: Proceedings of the 30th ACM Symposium on Access Control Models and Technologies, pp. 157–168 (2025)

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

[24] Xiang, C., Wu, Y., Shen, B., Shen, M., Huang, H., Xu, T., Zhou, Y., Moore, C., Jin, X., Sheng, T.: Towards continuous access control validation and forensics. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 113–129 (2019)

[25] Karimi, L., Aldairi, M., Joshi, J., Abdelhakim, M.: An automatic attribute-based access control policy extraction from access logs. IEEE Transactions on Dependable and Secure Computing **19**(4), 2304–2317 (2021)

[26] Ghavamnia, S., Palit, T., Benameur, A., Polychronakis, M.: Confine: Automated system call policy generation for container attack surface reduction. In: 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020), pp. 443–458 (2020)

[27] Li, X., Chen, Y., Lin, Z., Wang, X., Chen, J.H.: Automatic policy generation for {Inter-Service} access control of microservices. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 3971–3988 (2021)

[28] Zanatta, G., Caiazza, G., Ferrara, P., Negrini, L., White, R.: Automating ros2 security policies extraction through static analysis. In: 2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 3627–3634 (2024). IEEE

[29] Carrión, C.: Kubernetes as a standard container orchestrator-a bibliometric analysis. Journal of Grid Computing **20**(4), 42 (2022)

[30] Fischer, S., Urbanke, P., Ramler, R., Steidl, M., Felderer, M.: An overview of microservice-based systems used for evaluation in testing and monitoring: A systematic mapping study. In: Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024), pp. 182–192 (2024)