Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.062628

ARTICLE

Preventing IP Spoofing in Kubernetes Using eBPF

Absar Hussain[1], **Abdul Aziz**[1], **Hassan Jamil Syed**[2,*] **and Shoaib Raza**[1]

[1]FAST School of Computing, National University of Computer and Emerging Sciences, Karachi, 75030, Pakistan
[2]Asia Pacific University of Technology & Innovation (APU) Bukit Jalil, Kuala Lumpur, 57000, Malaysia

*Corresponding Author: Hassan Jamil Syed. Email: hassan.jamil@apu.edu.my

Received: 23 December 2024; Accepted: 10 April 2025; Published: 03 July 2025

ABSTRACT: Kubernetes has become the dominant container orchestration platform, with widespread adoption across industries. However, its default pod-to-pod communication mechanism introduces security vulnerabilities, particularly IP spoofing attacks. Attackers can exploit this weakness to impersonate legitimate pods, enabling unauthorized access, lateral movement, and large-scale Distributed Denial of Service (DDoS) attacks. Existing security mechanisms such as network policies and intrusion detection systems introduce latency and performance overhead, making them less effective in dynamic Kubernetes environments. This research presents PodCA, an eBPF-based security framework designed to detect and prevent IP spoofing in real time while minimizing performance impact. PodCA integrates with Kubernetes' Container Network Interface (CNI) and uses eBPF to monitor and validate packet metadata at the kernel level. It maintains a container network mapping table that tracks pod IP assignments, validates packet legitimacy before forwarding, and ensures network integrity. If an attack is detected, PodCA automatically blocks spoofed packets and, in cases of repeated attempts, terminates compromised pods to prevent further exploitation. Experimental evaluation on an AWS Kubernetes cluster demonstrates that PodCA detects and prevents spoofed packets with 100% accuracy. Additionally, resource consumption analysis reveals minimal overhead, with a CPU increase of only 2–3% per node and memory usage rising by 40–60 MB. These results highlight the effectiveness of eBPF in securing Kubernetes environments with low overhead, making it a scalable and efficient security solution for containerized applications.

KEYWORDS: CNCF; eBPF; pods; spoofing; IP; DDoS; container orchestration; packets; EKS; CNI; CNM; VM

1 Introduction

Containerization has revolutionized application deployment by enabling efficient resource management at the container level. Compared to traditional virtualization techniques such as EC2 instances, containerization offers a more lightweight, scalable, and efficient alternative for deploying applications. Containers are faster to deploy, recreate, and delete, which has led to their widespread adoption across various industries. This shift has resulted in the replacement of traditional virtual machines with containerized applications, offering enhanced scalability and management capabilities. As a result, numerous container orchestration tools, such as Kubernetes [1], Docker Swarm, AWS Elastic Kubernetes Service (EKS), Google Kubernetes Engine (GKE), and Azure Kubernetes Service (AKS), have been developed to manage containerized workloads efficiently.

Among these orchestration tools, Kubernetes has emerged as the most widely adopted solution [1,2]. Kubernetes provides a multi-layered architecture that enhances resource management, scalability, and security. At the lowest level, the architecture consists of the OS kernel, responsible for managing hardware

resources and isolating containers. Above the kernel layer lies the container runtime, which manages container instances and images. At the top layer, Kubernetes orchestrates and automates container management across multiple nodes [2].

The Container Network Interface (CNI) is the primary networking framework in Kubernetes, responsible for managing network resources and enabling cross-node communication. Various CNI plugins, such as Flannel, Calico, and Cilium, implement networking policies to enforce security. However, these mechanisms are susceptible to IP spoofing attacks, where an attacker forges a source IP address to impersonate a trusted entity, bypass access controls, or disrupt communication. CNI offers a more lightweight and flexible networking model that decouples container networking from the runtime. CNI is supported by multiple platforms, including Kubernetes, Apache Mesos, and OpenShift, making it the *de facto* standard for modern container networking. It enables IP address allocation, routing, and network policy enforcement without requiring a persistent daemon process. Kubernetes uses CNI to connect pods within clusters, ensuring efficient communication while providing the flexibility to integrate with external networking solutions such as Calico, Flannel, and Cilium [2,3].

Kubernetes networking is implemented through the Container Network Interface (CNI), which integrates various network plugins such as Flannel, Calico, and Cilium to manage connectivity within the cluster. The Kubernetes networking model must address four key communication challenges [4]:

1. Container-to-container communication within the same pod.
2. Pod-to-pod communication across different nodes.
3. Pod-to-service communication using service discovery and load balancing.
4. External-to-service communication to allow progress and egress traffic.

One of the main benefits of Kubernetes networking is that pods communicate directly via their assigned IP addresses, eliminating port mapping complexities. However, this architecture also introduces significant security concerns, particularly in multi-tenant environments, where multiple applications share the same underlying infrastructure [2,5].

According to the 2023 report by the Cloud Native Computing Foundation (CNCF), Kubernetes adoption has grown significantly, with its usage rising to 84% in 2023, up from 81% in 2022. Furthermore, 71% of organizations currently leverage container orchestration, while an additional 18% are in the evaluation phase [6].

While Kubernetes provides efficiency and scalability, it also introduces significant security vulnerabilities. Some of the most pressing concerns include [7]:

- IP Spoofing Attacks: Malicious actors can forge IP addresses to bypass access controls and impersonate trusted services.
- Lateral Movement Attacks: If one container is compromised, attackers can escalate privileges and access other services within the cluster.
- Configuration Mismanagement: Poorly defined network policies can expose sensitive services to unauthorized access.

A significant risk in Kubernetes security is IP spoofing, where attackers manipulate source IP addresses to gain unauthorized access. Many containerized environments still rely on IP-based access control mechanisms, such as iptables and network policies, which are vulnerable to spoofing techniques. Existing intrusion detection systems (IDS) and firewall mechanisms introduce latency and performance overhead, making it difficult to balance security and efficiency [5,7].

Given the limitations of static IP-based security mechanisms in Kubernetes, there is a critical need for a real-time, identity-based security model that can effectively prevent IP spoofing, lateral movement, and unauthorized service impersonation by leveraging Extended Berkeley Packet Filter (eBPF) technology to provide low-latency, kernel-level security enforcement, real-time packet validation, and dynamic network observability, ensuring enhanced resilience against advanced cyber threats while maintaining high performance and scalability in containerized environments.

To address this challenge, we propose a sandboxed security solution within the Linux kernel using eBPF, which operates directly on packet metadata without leaving kernel space. This approach enables deep packet inspection using Go-based eBPF programs and kernel-level network telemetry to verify node and packet attributes dynamically. Additionally, the proposed solution utilizes Linux kernel utilities to retrieve real-time information about active pods on each node, ensuring that incoming packets are authenticated based on their legitimate sources. By continuously validating source IP addresses at the kernel level, this mechanism effectively detects and prevents IP spoofing attacks in Kubernetes without introducing significant computational overhead or network latency.

To assess the efficiency and scalability of the proposed solution, we conducted extensive testing under high-load conditions and across multiple Kubernetes nodes. Detailed performance evaluation, latency analysis, and resource overhead measurements are presented in Section 5. We have summarized the contributions of this study as follows:

- Review and analysis of the state-of-the-art Container Network Interfaces (CNIs) and security tools aimed at preventing IP spoofing in Kubernetes.
- Identification and evaluation of the most efficient security technologies/tools available for IP spoofing prevention in containerized environments.
- Assessment of the performance overhead of the eBPF-based security mechanism, evaluating its feasibility and efficiency in real-world Kubernetes deployments.
- Development and implementation of an eBPF-based security solution designed to detect and mitigate IP spoofing attacks at the kernel level, ensuring real-time enforcement with minimal computational overhead.
- Quantification of latency and resource overhead introduced by the proposed solution to ensure optimal performance without significant impact on Kubernetes' workloads.

The remainder of this paper is structured as follows: Section 2 provides a comprehensive literature review, discussing existing research. Section 3 presents the proposed solution, detailing its design, functionality, and security mechanisms. Section 4 outlines the architectural framework of the proposed solution, explaining its components, and implementation details. Section 5 presents experimental results, including performance evaluation, latency analysis, and resource consumption metrics and Section 7 concludes the study.

2 Literature Review/Related Works

Kubernetes uses Container Network Interface (CNI). A CNI is a software interface between Container Runtime Interface (CRI) and network implementation. And deals with container network connectivity and by allocating and deleting resources when container gets created or deleted. There are multiple CNI plugins available and amongst the famous opensource are Calico and Cilium [6].

Researchers have extensively analyzed Project Calico's security and performance in Kubernetes environments. In a study evaluating network policies, Ref. [8] assessed the performance overheads of eBPF-based

solutions by Calico and Cilium. The findings indicated that Calico's implementation incurs negligible performance overhead, making it suitable for low-latency inter-container communication.

Further, States Usage & Adoption Report [9] highlighted that 35% of users prioritize Calico for its scalable networking and robust security policies. The report also noted that 85% of users implement network segmentation to protect east-west traffic, underscoring Calico's role in enhancing security within Kubernetes clusters. Calico has extended its network security capabilities to virtual machines and hosts, providing comprehensive protection across diverse environments. These enhancements include fine-tuned runtime threat detection and support for iptables, ensuring consistent performance and compatibility.

In summary, both empirical research and user reports affirm that Project Calico offers robust security features with minimal performance impact, making it a preferred choice for container networking and security in Kubernetes ecosystems [10].

In recent years, Cilium has garnered significant attention for its innovative approach to networking and security within cloud-native environments, primarily due to its utilization of eBPF (extended Berkeley Packet Filter) technology. Researchers have highlighted several key aspects of Cilium, particularly its impact on performance and security in containerized infrastructures [11].

One of the most notable performance benefits of Cilium is its ability to enhance networking efficiency through eBPF. As demonstrated by Isovalent, Cilium processes network packets directly within the Linux kernel, significantly reducing the need for user-space processing and thereby minimizing latency [11]. This integration enables Cilium to achieve high throughput and low latency in various scenarios, surpassing traditional networking solutions such as Calico. For example, benchmark tests show that Cilium outperforms Calico in both intra-node and inter-node communication, achieving throughput levels of up to 16 and 5 Gbps, respectively. Such improvements are particularly beneficial for environments with high network traffic, such as Kubernetes clusters [11,12].

Moreover, Cilium's use of eBPF has proven advantageous in service mesh scenarios. Research comparing the performance of service meshes, including Cilium, Istio, and Linkerd, reveals that Cilium maintains competitive performance, with latency increases observed when mutual TLS (mTLS) is enforced. Despite a 99% latency increase at 3200 requests per second (RPS), Cilium shows performance improvements as the request rate rises, suggesting that it can efficiently handle higher traffic loads [11]. This scalability makes Cilium a favorable choice for cloud-native applications that require high levels of secure communication.

From a security perspective, Cilium introduces several innovative features that enhance the security posture of cloud-native environments. One of the primary security strengths of Cilium is its support for fine-grained network policy enforcement. Cilium allows the definition of both Layer 3/4 and Layer 7 network policies, offering a more detailed and customizable approach to securing communications between services. This capability is essential for implementing a zero-trust security model, which is increasingly adopted in microservices architectures [11,12]. Additionally, Cilium provides transparent encryption through protocols like IPSec and WireGuard, which ensure that data in transit is secure without requiring changes to the application code. This feature simplifies the adoption of secure communication practices while maintaining high levels of performance. Furthermore, Cilium's runtime enforcement capabilities ensure that processes outside defined policies are prevented from running, further strengthening its security by ensuring that only authorized services and processes are permitted [13].

In conclusion, the integration of eBPF in Cilium not only provides significant performance advantages, particularly in terms of throughput and latency, but also enhances security by enabling comprehensive policy enforcement and secure communication practices. Researchers have found that Cilium's innovative use of kernel-level packet processing, coupled with its robust security features, makes it a compelling solution for

modern containerized environments [11,13]. As cloud-native technologies continue to evolve, Cilium's unique combination of performance and security positions it as a leading choice for organizations seeking efficient and secure networking solutions.

Bastion hosts are commonly employed as critical components in securing network infrastructures, particularly in enterprise and cloud environments. These hosts are specialized servers designed to act as intermediaries between an internal network and external networks, enhancing security by limiting direct access to sensitive systems. Researchers have extensively discussed the role of bastion hosts in improving network security, particularly in securing access to isolated or high-value systems, and the architectural considerations for their effective deployment.

Bastion hosts primarily serve as a gateway, providing a secure entry point into a network from external sources. The function of a bastion host is to mitigate the risks associated with exposing internal systems directly to the internet, thereby reducing the attack surface [14]. Bastions are typically placed in a demilitarized zone (DMZ) and act as a controlled interface for administrators or users to access the internal network. By routing all remote administration or SSH connections through the bastion, organizations can ensure that sensitive systems are not directly accessible, thereby preventing unauthorized access [15]. This security strategy is particularly important in environments that rely on multiple layers of network segmentation.

Several studies have highlighted the key security features of bastion hosts that contribute to the defense-in-depth strategy. According to [14], bastion hosts often incorporate multi-factor authentication (MFA) and logging mechanisms to track and monitor access attempts, making it easier to detect suspicious activities and prevent unauthorized access. MFA, when combined with secure tunneling protocols such as SSH or RDP, strengthens the authentication process, ensuring that only authorized users can access the internal network through the bastion [14,15]. Additionally, detailed logging of all interactions through the bastion host allows for comprehensive audit trails, which are crucial for both incident detection and post-event analysis.

In terms of architectural design, bastion hosts are typically configured with strict access controls, limiting the ports and services that are exposed to external connections. This ensures that only specific applications or management tools are accessible via bastion, further reducing potential entry points for malicious actors. Researchers have noted the importance of employing hardened bastion configurations to prevent exploitation through vulnerabilities [16]. This includes the use of minimalistic operating systems, disabling unnecessary services, and applying regular security updates. By adhering to these principles, bastion hosts can provide a highly secure entry point while minimizing the risk of compromise [17].

One significant advantage of bastion hosts is their scalability and adaptability in cloud environments. In cloud-native infrastructures, bastions can be dynamically deployed and scaled based on the security needs of the organization. For instance, cloud service providers such as AWS and Azure offer bastion host services that integrate with their existing security frameworks, providing seamless integration with identity and access management (IAM) policies [17]. These cloud-based bastions are particularly beneficial for organizations that require secure remote access for administrators or contractors without exposing sensitive internal systems.

Despite their benefits, researchers have pointed out that bastion hosts are not without limitations. A key challenge is ensuring that bastions are properly configured and monitored. If misconfigured or inadequately maintained, bastions could become a point of vulnerability, potentially allowing attackers to gain unauthorized access to the internal network [14]. Furthermore, the reliance on a single point of entry for remote access can create a bottleneck, especially in large organizations with multiple administrators and high-volume traffic [16]. Researchers recommend implementing additional layers of security, such as network segmentation and automated security scans, to mitigate these risks.

In conclusion, bastion hosts are a critical component of modern network security, providing a secure access point between external and internal networks. Their use of multi-factor authentication, strict access controls, and logging mechanisms enhance security, while their scalable nature makes them well-suited for both on-premises and cloud environments. However, to maximize their effectiveness, bastion hosts must be carefully configured and maintained, and organizations should consider supplementary security measures to address potential vulnerabilities [14,17].

Subaco [18,19] is a sandbox tailoring environment developed to address critical networking issues within containerized Platform as a Service (PaaS) environment. Specifically, it targets two main issues associated with containerized PaaS networking: (1) source verification of forged packets and (2) network isolation policy enforcement. These challenges are deeply rooted in the inherent complexities of networking within containerized and microservices architectures, where the traditional networking models often fail to provide sufficient security and isolation. The solutions offered by Subaco to resolve these issues include: (1) verifying the packet source using a combination of information beyond just the packet header, and (2) enforcing network isolation policies without relying on the host OS kernel, which can be vulnerable to compromise. These solutions aim to enhance the overall security of containerized environments by mitigating the risks of packet forgery and unauthorized access between isolated network segments. By integrating additional metadata from the Subaco hypervisor, it provides a more robust verification mechanism.

The packet source verification process in Subaco occurs in three stages, when a container is initialized, the container runtime requests the Subaco hypervisor to assign an IP address to the container's virtual NIC. This step ensures that the IP address allocation is secure and consistent with the container's context. As the container sends packets using its virtual NIC, a context switch occurs between the Subaco hypervisor and the host operating system. The packet verifier within the hypervisor then cross-checks the container's identity, ensuring that the packet is coming from a verified and authenticated source. The packet header is cross verified against the container's internal metadata, including its IP and MAC addresses. Subaco ensures that the packet's source IP and MAC address match the expected values, thus preventing any potential spoofing or unauthorized packet injections. By integrating the container's contextual metadata into the verification process, Subaco creates a multi-layered security mechanism that is more resilient against attacks like IP spoofing, which is a common vulnerability in containerized environments [6,20].

Another key challenge Subaco addresses is the enforcement of network isolation policies in containerized environments. Network isolation within containerized PaaS architectures is often managed by the host kernel, which presents a significant security risk because compromising the kernel can lead to the breach of container isolation. To resolve this, Subaco implements network isolation policies at the hypervisor level, independent of the host OS kernel. This ensures that even if the host OS is compromised, the network isolation between containers remains intact. The policy enforces strict boundaries, ensuring that containers can only communicate with the authorized segments of the network, thereby reducing the risk of lateral movement within the environment [18,21].

There have been some solutions to prevent IP spoofing like BASTION and Subaco. In our implementation, we utilize eBPF technology and integrate it with existing CNI plugins. In this way we can integrate and test the solution on currently deployed infrastructure. eBPF being quite lightweight with low CPU and Memory overhead is easily integrable without any requirement of specific resources and architecture. Moreover, PodCA takes the necessary steps and re-spins the pod if the spoofing continues in this way probable attacks are further reduced as number of packets that can be sent from a compromised container is quite low hence revoking the access of hacker from the pod where attacker might have penetrated. Following below Table 1, comparing Calico, Cilium, BASTION, and Subaco against PodCA, highlights their key features, advantages, disadvantages, and how PodCA offers innovative improvements.

Table 1: Comparative analysis of Kubernetes CNI and security solutions

Ref.	Technology	Key features	Advantages	Disadvantages	Comparison with PodCA
[22,23]	Calico	Layer 3 (IP-based) routing Uses Border Gateway Protocol (BGP). Enforces Kubernetes Network Policies.	Efficient routing without overlays. MAC address consistency prevents Layer 2 attacks. Global and namespaced policies support fine-grained control.	No deep packet inspection (DPI). Lacks built-in ARP spoofing prevention.	PodCA enhances Calico by integrating eBPF-based IP spoofing detection and mitigation, which Calico lacks. Unlike Calico, PodCA actively mitigates ongoing attacks by respawning compromised pods.
[24,25]	Cilium	eBPF-based security and observability Supports Layer 3–7 filtering Prevents ARP spoofing using direct ARP responses.	Highly efficient eBPF-based security. Supports Layer 7 filtering for deep packet inspection. Identity-based access control prevents IP spoofing.	Complex to deploy due to eBPF integration. Lacks full sandboxing for malicious containers.	PodCA improves upon Cilium by adding real-time pod respawning for attack mitigation, while Cilium only enforces policies. PodCA's low CPU/memory overhead makes it easier to deploy.
[25,26]	Bastion	Decentralized per-container network stack. Least privilege communication enforcement. Prevents unauthorized eavesdropping.	Strong isolation of container communication. Only interdependent containers can communicate Reduces lateral movement of attacks.	High CPU/memory overhead due to per-container network stack. Limited scalability in dynamic microservices.	PodCA outperforms BASTION by using eBPF-based security instead of a heavy per-container network stack, reducing resource overhead while maintaining strong isolation.

(Continued)

Table 1 (continued)

Ref.	Technology	Key features	Advantages	Disadvantages	Comparison with PodCA
[27]	Subaco	Sandboxing-based network security. Hypervisor-level packet verification. Kernel-independent network isolation.	Verifies packets beyond headers using hypervisor metadata. Does not rely on the host OS kernel for security. Blocks IP spoofing and prevents kernel exploits.	Higher network latency due to hypervisor-level checks. Not easily integratable with Kubernetes CNIs.	PodCA surpasses Subaco by implementing eBPF-based in-kernel security, avoiding hypervisor overhead and enabling faster attack mitigation. PodCA works seamlessly with existing CNIs.

PodCA represents a next-generation approach to Kubernetes network security by integrating eBPF-based attack mitigation with existing CNIs. Unlike Calico and Cilium, which focus on policy enforcement, and BASTION and Subaco, which introduce resource-intensive sandboxing, PodCA offers a lightweight, adaptive solution that actively mitigates network attacks by respawning compromised pods. PodCA minimizes security risks through real-time threat detection and automated remediation while ensuring high-performance networking in microservices architectures.

Extended Berkeley Packet Filter (eBPF) is a powerful and highly flexible technology that allows users to run custom programs within the kernel without modifying kernel source code. Initially developed for network packet filtering, eBPF has evolved to become a key enabler for modern security, performance monitoring, and networking solutions in containerized environments. Researchers have explored the various aspects of eBPF, examining its potential and real-world applications in different areas such as container security, network observability, and system monitoring [28].

In the context of containerized environments, eBPF has emerged as a critical tool for improving security. Researchers highlight its use for fine-grained security enforcement by monitoring and controlling system calls, network traffic, and container interactions at the kernel level. eBPF programs are dynamically loaded into the kernel to perform actions like packet filtering, system call interception, and enforcement of network policies without incurring significant performance penalties. One of the primary advantages of eBPF in container security is its ability to enforce network isolation and detect anomalous behaviors in real-time without relying on traditional security models that can add overhead or complexity [29].

One key area where eBPF enhances container security is by providing runtime system call filtering, which helps prevent malicious containers from performing unauthorized operations. This approach allows for the isolation of containerized applications and limits the potential attack surface [29].

eBPF has also become an essential technology for network monitoring and performance optimization in containerized environments. Researchers emphasize its ability to provide deep insights into network traffic with minimal overhead by attaching programs to various points in the networking stack. Unlike traditional network monitoring solutions, which may introduce significant latency or processing overhead,

eBPF programs operate directly within the kernel, allowing for real-time data collection and processing with negligible impact on system performance [30].

In containerized systems, eBPF can be used to capture and analyze network packets, providing a detailed view of communication between containers and services. This is especially useful in microservices architectures where traditional monitoring tools struggle to provide visibility into dynamic and ephemeral workloads. eBPF's ability to track network flows with low overhead makes it a suitable solution for performance-sensitive applications [30].

3 Proposed Solution and Its Architecture

Our proposed solution, PodCA, is designed based on the Container Network Interface (CNI) to enable seamless pod-to-pod communication without requiring a traditional network interface, thereby simplifying the communication process. However, it ensures security by preventing unauthorized and spoofed communication between pods. The key components of this architecture include:

3.1 Manager Integrated with CNI

As illustrated in Fig. 1, the PodCA architecture features a manager that maintains a global view of all pods within the network. This manager is responsible for tracking and storing pod mappings, ensuring efficient and secure communication while enforcing security policies.

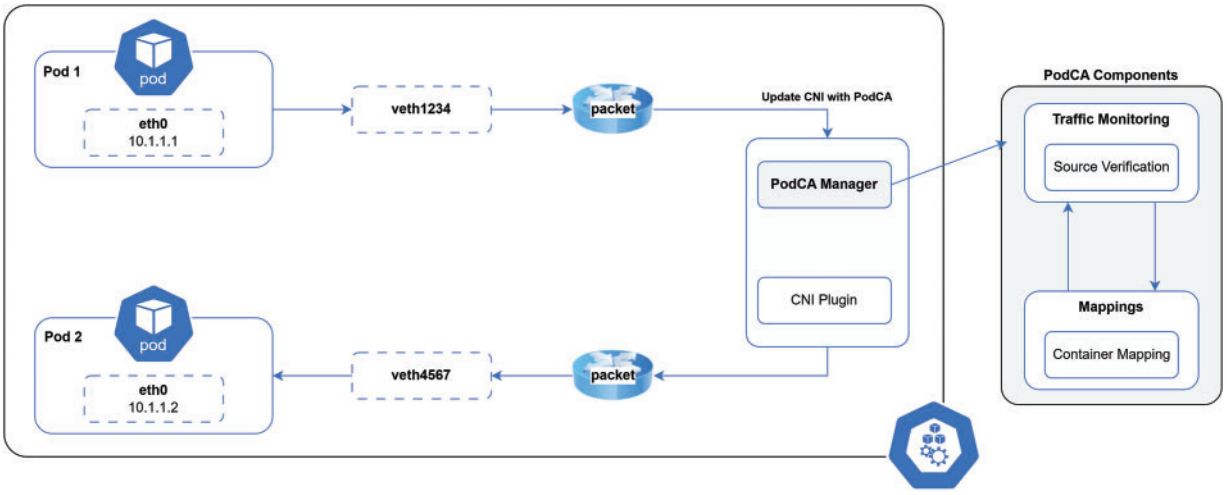

Figure 1: Architecture overview of PodCA

3.2 Container Network Map

PodCA maintains a container network map, associating virtual Ethernet interfaces (veth) with their assigned pod IP addresses. This mapping ensures accurate identification of containers within the network. The container network map plays a crucial role in facilitating secure pod-to-pod communication by verifying packet authenticity before transmission. Before forwarding a packet to its destination pod, PodCA cross-references the packet's details with the container network map. If the packet's source does not match an entry in the mapping, PodCA immediately drops the packet, preventing unauthorized or spoofed communication. This mechanism enhances security by ensuring that only legitimate packets are allowed to traverse the network.

3.3 CronJob

The CronJob is a scheduled task responsible for maintaining an up-to-date container network map by dynamically managing pod mappings, as depicted in Fig. 2. This process ensures that newly created pods are seamlessly integrated into the network map while simultaneously removing outdated or terminated pods. By automating this update mechanism, the CronJob preserves the accuracy of the network mapping, enabling PodCA to enforce security policies efficiently. This proactive approach prevents stale or unauthorized entries from lingering in the system, thereby enhancing the integrity and security of containerized environments. The mapping table is created and updated dynamically using the procedure described in Algorithm 1.

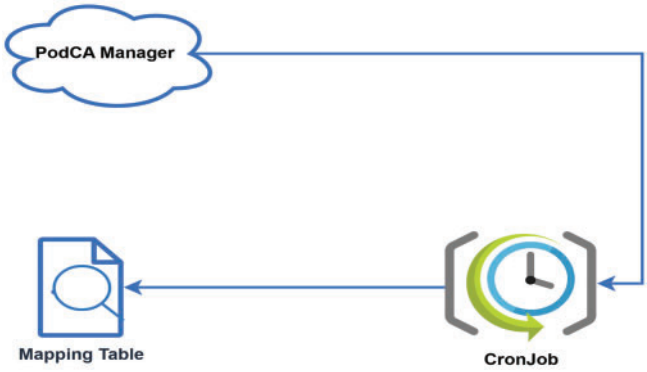

Figure 2: CronJob and mapping table

Algorithm 1: Constructing the mapping table for Kubernetes pods

Input:
P ← Set of all pods
M ← Initial mapping table of all pods
Found ← False
Output:
IPAddress, veth_interface
Ip_address ← IPAddress
veth_interface ← veth
For each container c ∈ P do
container_id ← containerid
container_set ← containerset
For each element m ∈ M do
If m [3] == Ip_address then
If m [0] == container_id then
Found ← True
If Found == False then
nm ← (container_id, container_set, ip_address, veth_interface)
AddItem(M, nm)

When packets reach the PodCA Manager, the virtual Ethernet (veth) interface captures them, extracting the source IP (src IP) for verification against the container network map. If the mapping is not found in the map, the program performs an individual lookup for the specific pod-to-IP association and proceeds

accordingly. If a valid mapping is confirmed, the packet is transmitted; otherwise, it is dropped to prevent unauthorized access.

3.4 DaemonSet

A DaemonSet ensures that a specific pod runs on every node within the cluster. PodCA can be deployed as a DaemonSet, enabling it to operate on each node and continuously collect network-related data to update the container network mapping. This deployment approach allows PodCA to seamlessly integrate with existing CNI plugins, providing an additional security layer for monitoring and enforcing network policies. By leveraging a DaemonSet, PodCA ensures real-time tracking of pod communication while maintaining compatibility with the existing infrastructure.

3.5 PodCA Manager

The PodCA Manager is responsible for two primary tasks:

1. Collecting Pod Information—It gathers detailed information about all pods within the cluster, including their network configurations, to build a comprehensive network topology.
2. Managing the Networking Stack—It oversees pod communication, ensuring security policies are enforced and unauthorized traffic is blocked.

3.6 Container Collection

The container collection process involves aggregating container-related data and constructing a global container network mapping, as shown in Table 2. PodCA leverages eBPF programs to collect vital networking information, including virtual Ethernet (veth) interfaces, IP addresses, container IDs, and associated network details. It then maps the veth interfaces of pods and nodes to establish a structured network view.

Table 2: Container network map

Container	Network	Interface	IP Address
Frontend1	Webservice	vethwepl6f964e8	10.109.122.53
Backend1	Webservice	vethweplb89dc35	10.99.35.135
Backend2	Webservice	vethweplb957e84	10.100.120.225
Database1	Webservice	vethweplc5ee33c	10.96.0.1

Additionally, PodCA creates a dependency map, identifying which containers can communicate with each other based on their associated services. Since pods are dynamic and can terminate or be recreated, the PodCA Manager periodically executes a CronJob to update the network mapping, ensuring accuracy and reflecting the latest state of the cluster. This dynamic mapping mechanism helps maintain security and visibility in a constantly evolving containerized environment.

PodCA is integrated with an eBPF-based network interface, enabling it to collect multi-node information efficiently. By deploying a DaemonSet on each node, PodCA gathers and updates pod mappings across multiple nodes in the cluster. This approach ensures that all containers within the Kubernetes cluster are stored in a unified mapping, allowing for seamless retrieval and utilization. The mapping contains comprehensive details, including node, pod, and container information, ensuring accurate and efficient network policy enforcement across the cluster.

4 Implementation

We implemented PodCA on Amazon Elastic Kubernetes Service (Amazon EKS, starting with the setup of a Kubernetes cluster. As part of the network configuration, we deployed Cilium as the CNI plugin to manage container networking.

The PodCA architecture is built on Kubernetes (AWS EKS), eBPF, and CNI plugins to ensure efficient network monitoring and security enforcement. A DaemonSet runs eBPF code on each node, monitoring network traffic in real time. The eBPF program is triggered using hooks whenever a packet arrives, allowing for efficient interception and analysis.

In PodCA's implementation, a gateway function is responsible for handling outbound traffic. The egress gateway first evaluates IP rules, then forwards packet details to a verification function. When a packet arrives, PodCA's processing function extracts key attributes, including the source IP, destination IP, and veth interface. The source IP is validated by cross-referencing the packet path with the mapping table, which maintains veth-to-IP associations.

The eBPF program attaches to network-related syscalls, specifically tcp_probe and ipv4.connect. These hooks enable the system to inspect every outgoing and incoming connection.

- eBPF retrieves the process ID (PID) and cgroup information associated with the packet.
- It maps the PID to a Kubernetes pod/container using cgroup v2 identifiers or through BPF maps that store container metadata.
- This ensures that the network traffic source is accurately linked to a specific pod rather than relying solely on traditional IP-based filtering.

The Algorithm 2 ensures that only authenticated packets are forwarded. If the verification is successful, the packet is transmitted to the destination pod, and its status is set to forward. However, if the packet details fail authentication.

Algorithm 2: Source IP validation for Kubernetes network security

```
Input:
    P ← Incoming packet
    M ← Mapping table
    PD ← Set of pods with assigned IPs
    validated ← False
1. Extract ip, container_id, veth from P
2. For each element m ∈ M do
3.         If (container_id, ip, container_set) ∈ m then
4.             validated ← True
5. If validated == False then
6.         For each pd ∈ PD do
7.             If pod_id == ip where pod_id ∈ PD then
8.                 If veth_interface == veth where veth ∈ PD then
9.                     validated ← True
10. If validated == True then
11.        Forward Packet
12. Else
13.        Drop Packet
```

PodCA drops the packet, marks the validation as false, and returns an error message to the egress gateway, identifying the source as a spoofed IP address.

PodCA employs real-time IP verification by monitoring TCP connection establishments and ensuring that the source IP aligns with its assigned pod identity. The verification process follows these steps:

- Tracking Connection Initiation:
  - The eBPF program hooks into the ipv4.connect syscall when a pod initiates an outbound connection.
  - It captures the source IP and compares it against the assigned IP of the originating pod.
  - If the source IP does not match the expected value from the pod's namespace, it is flagged as suspicious.
- Validation against Kubernetes Network Policies:
  - Using Kubernetes CNI metadata, PodCA verifies whether the source IP belongs to the same node or another pod within the cluster.
  - It checks whether the IP is spoofed from an external or internal attacker.
- Spoofing Detection via TCP Handshake Anomalies:
  - The eBPF program monitors SYN packets to detect discrepancies in handshake patterns.
  - If the TCP handshake lacks proper acknowledgment (ACK response), it suggests potential spoofing.
  - Additionally, the program tracks connection retries, unexpected RST (reset) packets, and IP fragmentation, which can indicate malicious activity.

Since PodCA enables verification of the actual source IP of a packet, all pod and packet details are accessible via the Hubble UI, where users can view comprehensive packet information.

When a spoofed packet is detected and dropped, PodCA extracts the veth interface or node IP and retrieves the corresponding pod details. These details are recorded in a table, and a spoofing attempt counter is incremented for the identified pod.

If a pod exceeds 10 spoofing attempts, PodCA automatically terminates it. If the pod is managed by a Kubernetes Deployment, a new pod is automatically spawned to replace it. However, if the pod was manually deployed, it remains terminated without replacement. This approach helps mitigate persistent spoofing attacks by dynamically enforcing network security at the container level.

5 Results

This section presents the empirical evaluation of the PodCA framework and highlights the key findings of our research. We begin by analyzing PodCA's performance based on the evaluated results. Specifically, we assess its packet filtering accuracy and overall efficiency. To achieve this, we measure accuracy by evaluating packet filtering performance across clusters of varying sizes. This section is structured into subheadings to provide a clear and detailed analysis of the experimental results, their interpretation, and the conclusions derived from the study.

5.1 Accuracy

Packet filtering accuracy is a critical feature of any security framework, ensuring its reliability for deployment in high-security environments. To validate the accuracy of PodCA, we conducted experiments across clusters of varying sizes. The results of these experiments are summarized in Table 3.

We tested the accuracy of our proposed PodCA solution across clusters ranging in size from 2 to 10 nodes. The results demonstrate that PodCA consistently achieved 100% efficiency in detecting and dropping spoofed packets, regardless of cluster size. These findings confirm that PodCA is highly reliable and effective in securing containerized environments.

Table 3: An overview of PodCA test experiment

eBPF hooks Tcp_probe>ipv4.connect (tcpstates-bpfcc)	Total connections	Spoofed requests	Legitimate requests	Spoofed traffic percentage	Detection efficiency
192.168.53.172	80	50	30	62.5%	100%
192.168.69.199	150	100	50	66.6%	100%
192.168.88.31	200	150	50	75%	100%
192.168.79.229	300	200	100	66.6%	100%

Additionally, we evaluated PodCA's prevention mechanism, specifically its ability to respawn pods in response to multiple spoofed packet attacks from a source pod. The system successfully triggered the pod respawn mechanism after detecting repeated malicious activity, further validating PodCA's proactive security measures.

To further illustrate the findings, Fig. 3 presents a comparative analysis of spoofed traffic percentage and detection efficiency across different source IPs. The graph demonstrates that despite variations in the percentage of spoofed traffic, PodCA consistently achieved 100% detection efficiency in all cases. This highlights PodCA's robust ability to identify and mitigate spoofed packet attacks effectively. The visualization reinforces the tabulated results from Table 3, confirming that PodCA maintains reliable security enforcement regardless of network conditions. The ability to accurately detect and drop spoofed packets ensures that malicious activities are effectively mitigated, making PodCA a highly efficient security framework for containerized environments.

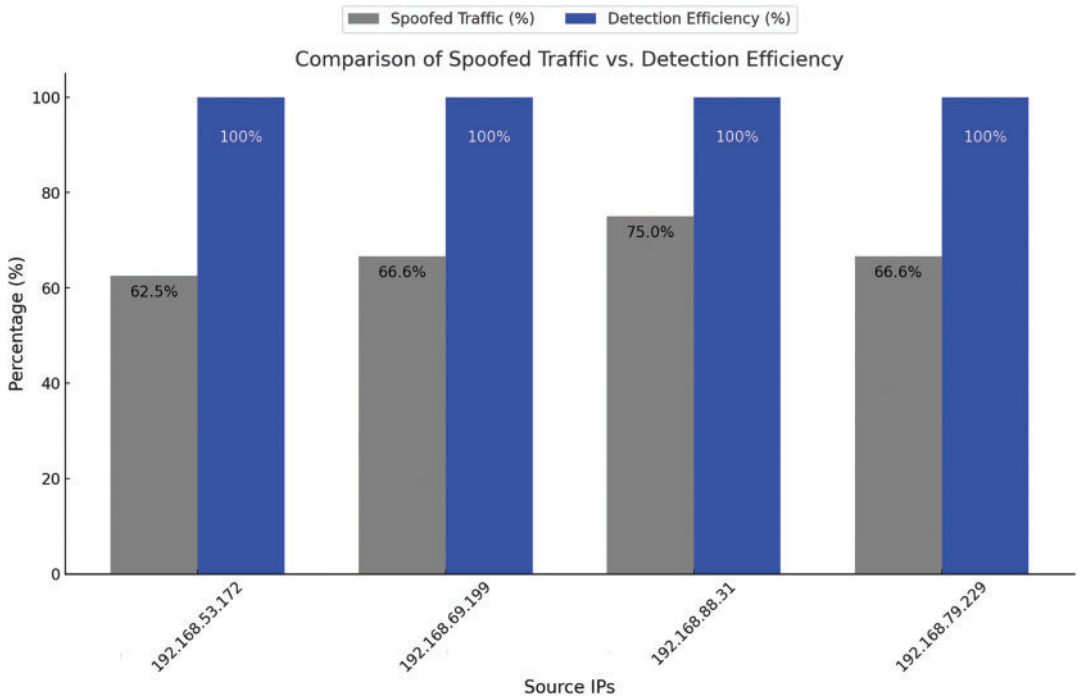

Figure 3: Detection efficiency across various source IPs

5.2 CPU Overhead

Another important resource to monitor in kubernetes is CPU. CPU and Memory are crucial in scheduling of pods and jobs on kubernetes high CPU can lead to high cost of nodes.

Here in order to monitor the CPU resources of the nodes we use CloudWatch metrics. Cloud watch metrics provide detailed overview of CPU of the nodes. The CPU metrics before and afer PodCA deployment are shown in Figs. 4 and 5.

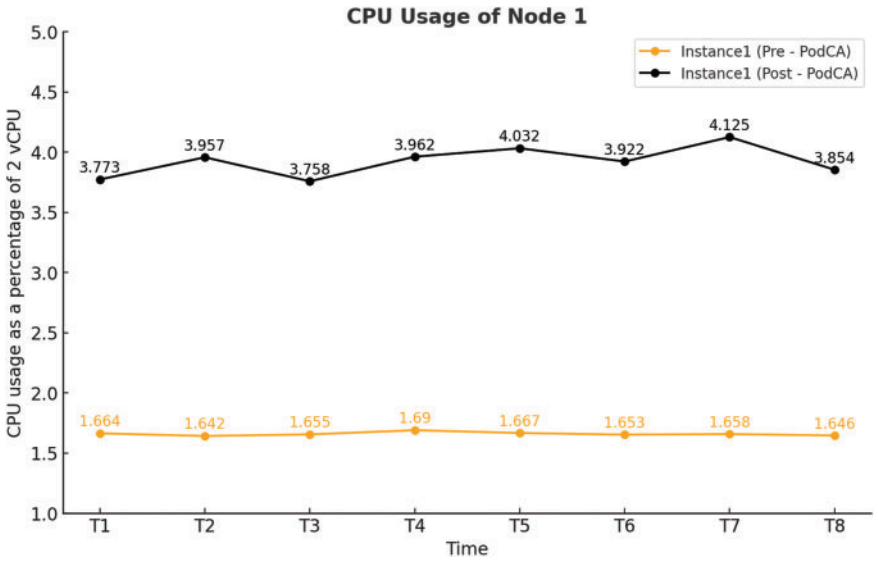

Figure 4: CPU usage of Node 1 before and after PodCA deployment

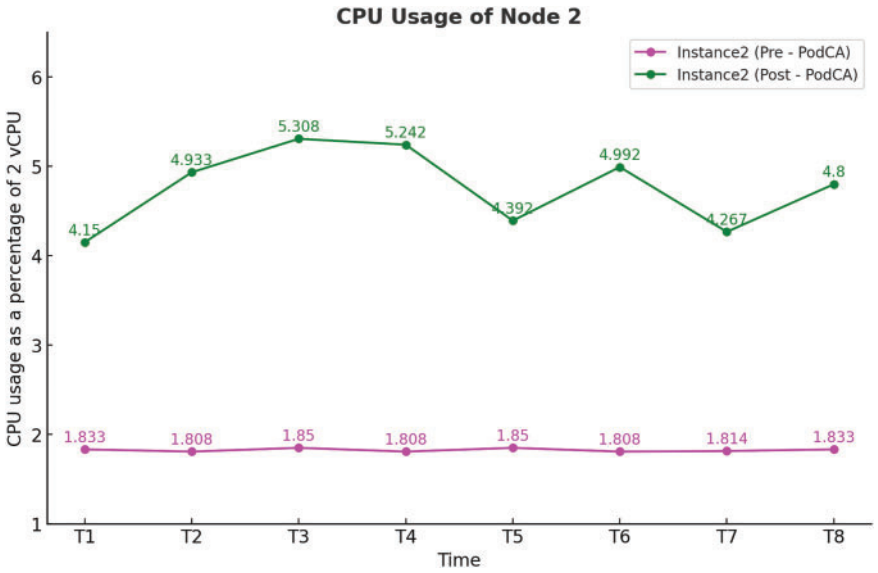

Figure 5: CPU usage of Node 2 before and after PodCA deployment

The graph for CPU before and after PodCA deployment show that there was minute CPU overhead for the two nodes. This CPU consumption is mainly caused by the CronJob that is deployed in order to have ConfigMaps for pod details and update the ConfigMaps one one minute bases. This CronJob slightly increases the CPU on node on which it is running.

5.3 Memory Overhead

This section analyzes the memory (RAM) overhead on the nodes pre and post deployment of PodCA. Figs. 6 and 7 illustrate the memory utilization trends for two nodes, measured as a percentage of 1 GB total RAM per node. Pre-deployment, Node 1 and Node 2 exhibited an average memory utilization of approximately 82% and 83%, respectively. Post-deployment, memory usage increased, with Node 2 experiencing a higher overhead than Node 1.

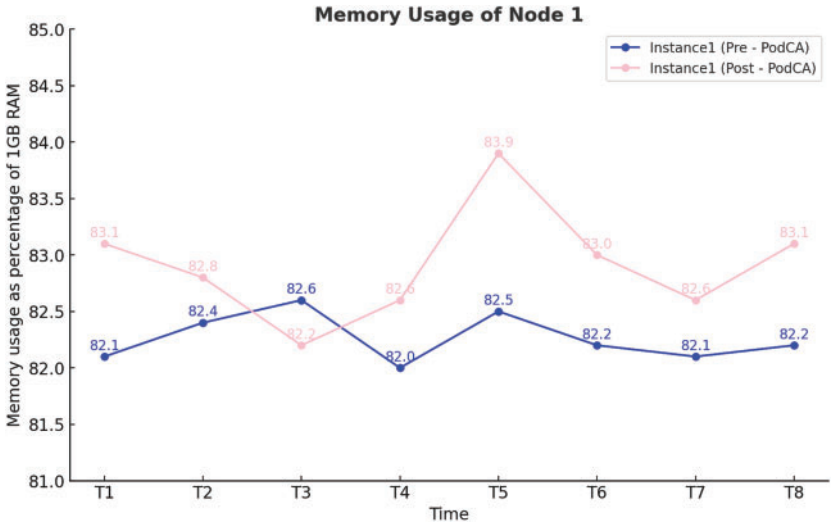

Figure 6: Memory Usage of Node 1 before and after PodCA deployment

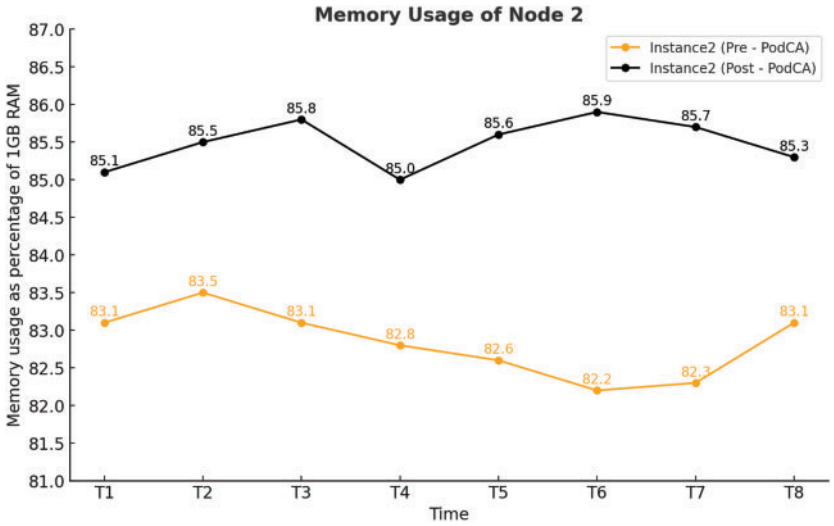

Figure 7: Memory Usage of Node 2 before and after PodCA deployment

The graphs clearly indicate that the increase in memory usage is primarily attributed to the CronJob scheduling mechanism rather than the core PodCA functions. The additional overhead from PodCA's actual security enforcement and packet filtering operations contributed only about 1% of total RAM usage. This suggests that PodCA's impact on system resources is minimal, making it an efficient and lightweight security solution for containerized environments.

6 Discussion

The implementation of PodCA for IP spoofing detection and prevention was evaluated across multiple scenarios, including different namespaces and cluster sizes. The results demonstrated 100% efficiency in detecting and mitigating forged source IP packets, ensuring that all unauthorized traffic was successfully dropped. This highlights PodCA's robustness in maintaining the integrity of container network security.

Furthermore, we compared PodCA with existing networking solutions such as Cilium, Calico, Flannel, and Docker, as summarized in Table 4. Unlike these solutions, which either lack IP spoofing detection mechanisms or rely on policy-based configurations, PodCA offers a comprehensive approach by both detecting and preventing such attacks autonomously.

Table 4: Success (✓): always Detects/Prevents. Failure (✗): does not detects or prevents IP spoofing. Conditional (▲): attack can be detected by policy enforcement

Feature	Cilium	Calico	Flannel	Docker	PodCA
Detects IP Spoofing	▲	✗	✗	✗	✓
Prevents/Countermeasures IP Spoofing	✗	✗	✗	✗	✓

In terms of resource consumption, the deployment of PodCA introduced a slight increase in memory and CPU utilization. Each node experienced an additional memory overhead of approximately 20–30 MB, leading to a total consumption of 40–60 MB per node. The CPU usage increased by around 2%–3% per node on a 2 vCPU system, equivalent to an overhead of 40–50 millicores per node, totaling 80–100 millicores across nodes. These results indicate that while PodCA adds a minor resource overhead, it remains a lightweight and efficient security solution for Kubernetes clusters.

Additionally, further analysis showed that the primary contributor to memory overhead was the CronJob scheduling mechanism, while the actual security enforcement and packet filtering operations contributed only 1% of total RAM usage. This suggests that PodCA's impact on system resources is minimal, making it highly suitable for deployment in large-scale containerized environments.

Overall, the findings confirm that PodCA is an effective and resource-efficient solution for securing containerized workloads against IP spoofing attacks, outperforming existing networking frameworks by providing real-time detection and proactive prevention mechanisms. Future improvements can focus on further optimizing resource utilization and expanding security policies to address additional network threats.

7 Conclusion

In conclusion, detecting and preventing IP spoofing in Kubernetes using eBPF presents a highly effective strategy to reduce the attack surface and mitigate security risks within containerized environments. As the adoption of Kubernetes continues to grow, the risk of IP spoofing attacks originating from compromised containers also rises. Such threats necessitate robust prevention mechanisms to safeguard Kubernetes-based

systems. By implementing an eBPF-based solution, real-time monitoring and mitigation of IP spoofing attacks become feasible, effectively reducing their impact before the entire system is compromised.

The system's response time and overall effectiveness can be further enhanced through an automated process that removes compromised pods and spins up new, secure pods. This automated response ensures a quicker security reaction, minimizes damage, and improves the uptime and availability of the system.

As discussed in Section 6, eBPF introduces minimal overhead, making it suitable for deployment in various environments without the need for specialized resources or modifications to existing infrastructure. This makes eBPF an attractive choice for network security in Kubernetes environments.

Our proposed solution, PodCA, for preventing IP spoofing in Kubernetes using eBPF, was successfully implemented in the AWS Kubernetes service (EKS). Initially, we deployed a CronJob to gather details about the pods running on each node. We then evaluated these details using Go Lang functions running on each node. Verified packets are forwarded to the destination pod, while spoofed packets are dropped, as detailed in Section 5.

The effectiveness of PodCA was observed on AWS EKS, where it efficiently handled incoming packets, dropping spoofed ones. Our observations confirm that eBPF incurs very low CPU and memory overhead, as highlighted in Section 6. PodCA operates efficiently, ensuring that network traffic is thoroughly validated before forwarding.

While PodCA is effective in detecting and blocking IP spoofing, it does have certain limitations. Specifically, it requires modifications to packet headers to prevent 100% of IP spoofing across all types of IPs. However, it does not require packet modification for preventing spoofing of internal pod IPs.

Overall, the use of eBPF for preventing IP spoofing in Kubernetes proves to be a highly efficient, low-overhead approach that can significantly improve security, reliability, and availability of systems at scale. By reducing the attack surface of containerized applications, this solution contributes to building a more resilient and secure Kubernetes infrastructure.

Acknowledgement: The authors would like to express their gratitude to FAST School of Computing, National University of Computer and Emerging Sciences, Karachi, Pakistan for their support and resources provided during this research.

Funding Statement: This research was partially supported by Asia Pacific University of Technology & Innovation (APU) Bukit Jalil, Kuala Lumpur, Malaysia. The funding body had no role in the study design, data collection, analysis, interpretation, or writing of the manuscript.

Author Contributions: Absar Hussain: Contributed to conceptualization, data collection, and initial manuscript drafting. Abdul Aziz: Involved in methodology design, data analysis, and manuscript review and editing. Hassan Jamil Syed: Led project supervision, study validation, and final manuscript revision. Shoaib Raza: Assisted in software implementation, result interpretation, and figure and table preparation. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: Not applicable.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

Glossary

CNCF	Cloud Native Computing Foundation
eBPF	Extended Berkeley Packet Filter
CNI	Container Network Interface
CRI	Container Runtime Interface
CNM	Container Network Model

References

1. German K, Ponomareva O. An overview of container security in a kubernetes cluster. In: Proceedings of the 2023 IEEE Ural-Siberian Conference on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT); 2013 May 15–17; Yekaterinburg, Russian. p. 283–5. doi:10.1109/USBEREIT58508.2023.10158865.
2. Kim B, Kim J, Lee S. Exploring security enhancements in Kubernetes CNI: a deep dive into network policies. IEEE Access. 2025;13(5):2169–3536. doi:10.1109/ACCESS.2025.3543841.
3. Scano D, Giorgetti A, Paolucci F, Sgambelluri A, Chammanara J, Rothman J, et al. Enabling P4 network telemetry in edge micro data centers with kubernetes orchestration. IEEE Access. 2023;11(1):22637–53. doi:10.1109/ACCESS.2023.3249105.
4. Burns B, Grant B, Oppenheimer D, Brewer E, Wilkes J. Kubernetes. Commun ACM. 2016;59(5):50–7. doi:10.1145/2890784.
5. Soldani D, Nahi P, Bour H, Jafarizadeh S, Soliman MF, Giovanna D, et al. eBPF: a new approach to cloud-native observability, networking and security for current (5G) and future mobile networks (6G and Beyond). IEEE Access. 2023;11:57174–202. doi:10.1109/ACCESS.2023.3281480.
6. CNCF Annual Survey 2023 [Internet]. [cited 2025 Apr 9]. Available from: https://www.cncf.io/reports/cncf-annual-survey-2023/.
7. Nam J, Lee S, Porras P, Yegneswaran V, Shin S. Secure inter-container communications using XDP/eBPF. IEEE/ACM Trans Netw. 2023;31(2):934–47. doi:10.1109/TNET.2022.3206781.
8. Dakić V, Redžepagić J, Bašić M, Žgrablić L. Performance and latency efficiency evaluation of kubernetes container network interfaces for built-in and custom tuned profiles. Electronics. 2024;13(19):3972. doi:10.3390/electronics13193972.
9. Ferguson L. Tigera Closes Out 2023 with Significant Momentum for Calico as Demand for Container Security Accelerates, Tigera—creator of Calico [Internet]. [cited 2025 Apr 9]. Available from: https://www.tigera.io/blog/tigera-closes-out-2023-with-significant-momentum-for-calico-as-demand-for-container-security-accelerates/.
10. Tigera. Tigera enhances calico with major network and runtime security updates [Internet]. [cited 2025 Apr 9]. Available from: https://www.prnewswire.com/news-releases/tigera-enhances-calico-with-major-network-and-runtime-security-updates-302301572.html.
11. Cilium netkit: the final frontier in container networking performance [Internet]. [cited 2025 Apr 9]. Available from: https://isovalent.com/blog/post/cilium-netkit-a-new-container-networking-paradigm-for-the-ai-era/.
12. Budigiri G, Baumann C, Mühlberg JT, Truyen E, Joosen W. Network policies in kubernetes: performance evaluation and security analysis. In: Proceedings of the 2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit); 2021 Jun 8–11; Porto, Portugal. p. 407–12. doi:10.1109/EuCNC/6GSummit51104.2021.9482526.
13. Cilium (computing) [Internet]. [cited 2025 Jan 1]. Available from: https://en.wikipedia.org/w/index.php?title=Cilium_(computing)&oldid=1262067602.
14. The crucial role of bastion hosts in securing your network infrastructure [Internet]. [cited 2025 Apr 9]. Available from: https://www.cloudthat.com/resources/blog/the-crucial-role-of-bastion-hosts-in-securing-your-network-infrastructure.
15. Teleport. 14 best practices to secure SSH Bastion Host [Internet]. [cited 2025 Apr 9]. Available from: https://goteleport.com/blog/security-hardening-ssh-bastion-best-practices/.
16. What is a Bastion Host and Does Your Business Need It [Internet]? [cited 2025 Apr 9]. Available from: https://nordlayer.com/blog/bastion-host/.

17. Vijayababu G, Haritha D, Prasad RS. An effective utilization of bastion host services in cloud environment. Int J Innov Technol Explor Eng. 2019;8(7):2215–20.
18. Wan Z, Lo D, Xia X, Cai L. Practical and effective sandboxing for Linux containers. Empir Softw Eng. 2019;24(6):4034–70. doi:10.1007/s10664-019-09737-2.
19. Jarkas O, Ko R, Dong N, Mahmud R. A container security survey: exploits, attacks, and defenses. ACM Comput Surv. 2025;57(7):1–36. doi:10.1145/3715001.
20. Mainas C, Plakas I, Ntoutsos G, Nanos A. Sandboxing functions for efficient and secure multi-tenant serverless deployments. In: Proceedings of the 2nd Workshop on SErverless Systems, Applications and Methodologies; 2024 April 22; Athens, Greece. New York, NY, USA: Association for Computing Machinery; 2024. p. 25–31. doi:10.1145/3642977.3652096.
21. Khalimov A, Benahmed S, Hussain R, Kazmi SA, Oracevic A, Hussain F, et al. Container-based sandboxes for malware analysis: a compromise worth considering. In: Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing; 2019 Dec 2–5; Auckland, New Zealand. New York, NY, USA: Association for Computing Machinery; 2019. p. 219–27. doi:10.1145/3344341.3368810.
22. Project Calico. Tigera—creator of Calico [Internet]. [cited 2025 Apr 9]. Available from: https://www.tigera.io/project-calico/.
23. Network Plugins. Kubernetes [Internet]. [cited 2025 Apr 9]. Available from: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/.
24. Native C-C, Networking EP-B. Observability, Security [Internet]. [cited 2025 Apr 9]. Available from: https://cilium.io/.
25. Theodoropoulos T, Rosa L, Benzaid C, Gray P, Marin E, Makris A, et al. Security in cloud-native services: a survey. J Cybersecur Priv. 2023;3(4):758–93. doi:10.3390/jcp3040034.
26. Nam J, Lee S, Seo H, Porras P, Yegneswaran V. BASTION: a Security Enforcement Network Stack for Container Networks. In: Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20); 2020 Jul 15–17; Virtual. p. 81–95.
27. Nakata Y, Matsubara K, Matsumoto R. Concentrated isolation for container networks toward application-aware sandbox tailoring. In: Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing; 2021 Dec 6–9; Leicester, UK. New York, NY, USA: Association for Computing Machinery; 2021. p. 1–10. doi:10.1145/3468737.3494092.
28. Unveiling eBPF: revolutionizing security and observability | Wiz Blog, wiz.io [Internet]. [cited 2025 Apr 9]. Available from: https://www.wiz.io/blog/unveiling-ebpf-revolutionizing-security-and-observability.
29. Sharaf H, Ahmad I, Dimitriou T. Extended berkeley packet filter: an application perspective. IEEE Access. 2022;10:126370–93. doi:10.1109/ACCESS.2022.3226269.
30. Findlay W. Security applications of extended BPF under the Linux Kernel [Internet].[cited 2025 Apr 9]. Available from: https://www.cisl.carleton.ca/~will/written/findlay20bpfsec.pdf.