# HELIOS: Hardware-assisted High-performance Security Extension for Cloud Networking

Myoungsung You
KAIST
famous77@kaist.ac.kr

Jaehyun Nam
Dankook University
jaehyun.nam@dankook.ac.kr

Minjae Seo
KAIST
ms4060@kaist.ac.kr

Seungwon Shin
KAIST
claude@kaist.ac.kr

## ABSTRACT

With the increasing adoption of containerization in cloud services, container networking has become a critical concern, as it enables the agile deployment of microservices but also introduces new vulnerabilities susceptible to network attacks, posing a threat to container environments. While several security solutions have been introduced to address this concern, they unfortunately exhibit significant shortcomings, including security vulnerabilities and limited performance. We thus propose *Helios*, a novel hardware-based network security extension that addresses the security and performance limitations in existing solutions. Leveraging a smartNIC, *Helios* enhances both the security and performance facets of container networking through two key mechanisms: (i) the establishment of physically isolated container communication channels and (ii) the network security engines fully offloaded to the smartNIC. Our evaluation shows that *Helios* mitigates various network threats initiated from both container- and host-side while performing up to 3x faster than the existing solutions in container communication.

## CCS CONCEPTS

• **Networks** → **Cloud computing**.

## KEYWORDS

Container Network, Security Policy Enforcement, SmartNIC


*SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA*

ACM ISBN 979-8-4007-0387-4/23/11…$15.00
https://doi.org/10.1145/3620678.3624786

**ACM Reference Format:**
Myoungsung You, Jaehyun Nam, Minjae Seo, and Seungwon Shin. 2023. HELIOS: Hardware-assisted High-performance Security Extension for Cloud Networking. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620678.3624786

## 1 INTRODUCTION

Containers, lightweight and isolated execution environments, are widely adopted as an alternative to virtual machines. As containers are suitable for microservices architecture in virtue of agile deployment and better resource utilization, most cloud service providers such as Amazon Web Services [10], Microsoft Azure [35], and Google Cloud Platform [25] have actively employed container technology. However, the increased use of containers has also led to an increase in container security incidents. A recent survey [8] found that 94% of respondents reported experiencing at least one container security incident in 2021. These incidents are often attributed to the shared kernel model of container systems, which allows a malicious container to access the host kernel and compromise other containers on the same host. Hence, the focus on container security has predominantly centered on enhancing the security of individual containers, encouraging the development of various solutions [11, 51, 52, 57, 70], such as Confine [52], to prevent malicious system access initiated by a single container.

However, simply fortifying security measures within individual containers is not sufficient to effectively address the security concerns arising from the connectivity between containers, which is the foundation of microservices distributed across multiple hosts. Thus, these security implications associated with the network connectivity between containers have emerged as a new attack vector, potentially exploitable to progressively compromise the entire cloud system. One such attack is lateral movement, in which an attacker who has hijacked one container can move within the cloud system by compromising neighboring containers through several

network attacks [65, 69]. Lateral movement, a key strategy in APT attacks, enables an attacker to penetrate deeply into the cloud system and extract sensitive data [9, 33].

To mitigate such threats, container network security solutions [15, 43, 44, 65] have adopted network access control, restricting unauthorized communication between containers. They utilize a security proxy to inspect the traffic transmitted from individual containers. However, this proxy-based approach suffers from significant challenges pertaining to both security and performance. First, in today's container network architecture, container traffic is routed via virtual interfaces situated within the host network stack. This lack of isolation between container communication and the host system exposes all container traffic to the host system, leaving it vulnerable to traffic eavesdropping or manipulation by a host-side attacker. Unfortunately, the proxy-based approach lacks security measures to safeguard the exposed container communication channel from such network threats.

Second, ensuring restriction over the application layer (L7) container communication (e.g., REST APIs) is crucial in the context of microservices. However, existing solutions have a performance limitation in securing L7 container communication. This limitation arises from their reliance on proxy-based access control, necessitating the inspection of payloads sent by numerous containers solely using software proxies. Consequently, this limitation imposes a considerable resource overhead and performance degradation of microservices.

**Our approach.** We first tackle such network security challenges stemming from the absence of network isolation in container environments by examining the potential attack vectors intrinsic to the container network architecture. Next, we illuminate the inherent constraints of existing solutions in mitigating these attack vectors with real-world examples. We then introduce *Helios*, a novel hardware-assisted network security extension for containers. This work is, to the best of our knowledge, the first aiming to establish a hardware-isolated container networking architecture.

*Helios* aims to enhance container network security by establishing physically isolated communication channels through a *Helios* secure bridge – a hardware bridge that maintains physical isolation from the host side, effectively thwarting container traffic exposure and malicious traffic injection from the host side. In addition, *Helios* enables high-performance security enforcement for these isolated channels by leveraging two hardware security engines: the flow and payload inspectors. These engines restrict container connections and thwart malicious access over L3 to L7 protocols by examining the packets passing through the isolated *Helios* entirely offloads these security measures to the smartNIC, offering superior performance than software solutions and ensuring that host system resources are preserved for efficient microservice operation. Furthermore, *Helios* guarantees the identity of all packets sent from containers, thereby safeguarding containers against unauthorized traffic inflow.

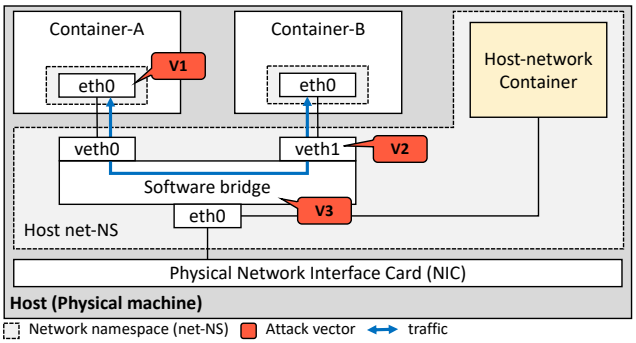



**Figure 1: Container networking architecture with potential network attack vectors, which a malicious container can exploit to compromise other containers.**

We implement a full prototype of *Helios* on a commodity ASIC-based smartNIC [36] that is widely found in modern cloud environments. Our evaluation shows that *Helios* facilitates physical network isolation among containers, mitigating a broad spectrum of network attacks within Kubernetes environments [30], while outperforming the state-of-the-art up to a 3x margin in terms of security enforcement.

**Contribution**. We make the following contributions.

- *Analysis of today's container networking architecture.* We extensively explore attack vectors within the container networking architecture and evaluate the limitations of existing solutions aiming to mitigate these attack vectors.
- *Design and implementation of Helios.* We propose *Helios*, a novel hardware-assisted solution that provides physically isolated container communication channels to enhance network security while ensuring traffic over these isolated channels is protected through hardware-offloaded security policy enforcement engines.
- *Evaluation of Helios with real-world examples.* We demonstrate the effectiveness and efficiency of *Helios* in protecting containers against a diverse range of network threats, highlighting its ability to outperform existing solutions by up to 3x in security policy enforcement.

## 2 BACKGROUND AND MOTIVATION

### 2.1 Container Networking Architecture

**Network Namespace.** Figure 1 illustrates the common container networking architecture found in widely used container platforms such as Docker [18] and Kubernetes [30]. Each container operates within its discrete network namespace [2] (*net-NS*) and possesses dedicated network interfaces (e.g., eth0 within a container). Consequently, this leads to a

logical isolation of containers, with no visibility or accessibility to the network interfaces or traffic of other containers. The host system also maintains a network namespace (referred to as *Host net-NS* in Figure 1), encompassing network interfaces (e.g., eth0 in the host) connected to the physical network interface cards (NICs) of the host system.

**Bridge Network.** In default configurations, container platforms leverage a bridge network to enable inter-container connectivity, as containers cannot directly communicate due to net-NS isolation. Hence, as illustrated in Figure 1, containers are connected to a software bridge (e.g., Linux bridge [12]) via host-side interfaces [45] (e.g., veth0 in the host net-NS), paired with their container-side counterparts. The software bridge and host-side interfaces then enable inter-container communication while also facilitating the connections between containers and external networks.

**Host-Network Container.** Container platforms support a unique container known as a host-network container. This container does not have a distinct net-NS but instead shares the net-NS of the host system [19], directly accessing and using the host system's network interfaces and associated IP addresses. The use of host-network containers spans across a wide spectrum of use cases. Host-network containers are typically leveraged for performance optimization [28, 71], as they eliminate the overheads introduced by Network Address Translation (NAT) and routing through a software bridge. They are also utilized for direct service exposure to external networks (e.g., load balancers [26], reverse proxies [20], and VPN services [38]). Additionally, these containers are a viable solution for deploying legacy applications [21], which are designed to operate on a host network and bind specific ports on the host system. Furthermore, they are instrumental in executing network management services [13, 16, 22, 24] within container platforms. Note that although host-network containers can use the network interfaces of the host system, they do not gain privileged access to other resources of the host system. They cannot, for example, modify or disable a network interface unless they are granted.

## 2.2 Challenges in Container Networks

Prior research [61, 65, 69] has primarily focused on mitigating the attack vector originating from within a container (V1). However, it fails to consider that container traffic can become exposed to the host system due to the interconnectedness of the communication channel for containers and the host system, despite net-NS isolation. As a result, this exposure paves the way for the creation of new, critical network attack vectors that enable an attacker to laterally migrate to other containers within the host system. Here, we discuss three primary network attack vectors as depicted in Figure 1.

**Container-side Interface (V1).** To gain access to a container, an attacker first compromises a containerized service exposed to the Internet. Upon gaining control of this compromised container, the attacker can leverage the network interfaces within the container to launch lateral attacks against other accessible containers using various techniques such as container discovery, port scanning, and malicious traffic injection via the container's network interface [56, 65]. For instance, the attacker may manipulate DNS responses sent to target containers, steering them to interact with a malicious server (i.e., DNS spoofing), even without direct access to the DNS requests of the target containers. The attacker may also reroute the network traffic originating from target containers to the compromised ones using ARP spoofing, setting the stage for subsequent phases of the attack kill chain.

**Host-side Interface (V2).** Consider a situation where an attacker compromises a host-network container exposed to the Internet instead of regular containers. In such a scenario, the attacker gains more expansive network visibility to all network interfaces within the host system, including the host-side interfaces of running containers (e.g., *veth0* and *veth1* in Figure 1), regardless of net-NS isolation. Consequently, the attacker can directly monitor network traffic flowing to all running containers using traffic capture tools (e.g., tcpdump[42]) and inject fabricated packets into target containers to initiate lateral attacks (e.g., DNS spoofing and TCP session hijacking) without any restrictions. In short, whereas an attacker with V1 can only exploit a network interface and traffic associated with the compromised container, one with V2 can manipulate network interfaces and traffic for all running containers on the host system.

**Software Bridge (V3).** The software bridge, by virtue of routing all network traffic of containers, emerges as an attractive target for an attacker aiming to gain comprehensive visibility into the entire container network. While an attacker with both V1 and V2 can launch network attacks targeting a specific container, the use of the bridge broadens this scope, enabling simultaneous attacks against multiple containers. For instance, by leveraging the bridge, the attacker can intercept traffic flowing between all containers within the host, using traffic analysis techniques [62, 68, 73] to discern connectivity patterns and communication dependencies, and to extract sensitive data from the intercepted traffic.

## 2.3 Limitations of Contemporary Solutions

Several network security solutions [15, 39, 43, 44, 65] have been developed to enhance the security of container environments. They typically utilize security proxies that examine container traffic flow, enforcing security policies. However, this approach has significant limitations in both security and performance aspects when faced with network attack vectors

**Table 1: The effectiveness of existing security solutions from the security and performance perspectives. ✔ :can mitigate, ▲ :can partially mitigate, ✖ :cannot mitigate attack vectors.**

| *Security Solutions* | *Deployment* | *Attack Vectors* | | | *Performance* | |
|---|---|---|---|---|---|---|
| | | *V1* | *V2* | *V3* | *L3-4* | *L7* |
| Kernel-level proxy [15, 39, 64, 65] | Host net-NS | ✔ | ✖ | ▲ | High | N/A |
| User-level proxy [43, 44] | Container net-NS | ✔ | ✖ | ✖ | N/A | Low |
| *Helios* | Hardware | ✔ | ✔ | ✔ | High | High |

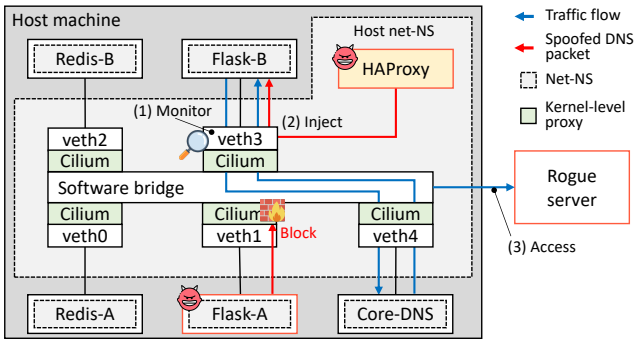



**Figure 2: A motivating example environment. An attacker conducts DNS spoofing against the victim (Flask-B) from two different locations (Flask-A and HAProxy).**

endemic to container networks. We delve into the limitations of these solutions with a summarization presented in Table 1.

**Kernel-level Proxy.** Cilium [15], Calico [39], and Bastion [65] represent contemporary solutions that enable security policy enforcement for container traffic. These solutions, effectively restricting unauthorized network access from the container side, employ a kernel-level proxy implemented using recent kernel networking features (e.g., eBPF [23]), which operates within the host net-NS. This proxy scrutinizes outgoing traffic from each container's net-NS (V1) and then discards any packets not conforming to the security policies or those identified as spoofed before they reach the software bridge. It subsequently redirects the filtered traffic to the host-side interfaces of the destination containers.

However, these solutions face certain limitations, especially when dealing with malicious packet injections originating from host-side interfaces (V2). The fundamental issue resides in the fact that these injections originate from the host net-NS, rather than a distinct container net-NS, hence existing measures struggle to intercept and scrutinize such packets effectively. Moreover, despite their efforts to reduce traffic exposure and partially mitigate V3 with direct traffic redirection, they fall short of preventing an attacker from directly injecting malicious packets into the bridge (V3) with the same reason as V2. Consequently, malicious packets injected from the host net-NS can circumvent these solutions and access target containers without any restrictions.

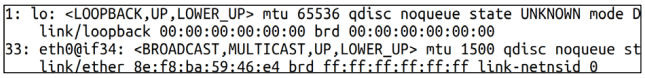
Another key limitation of these solutions lies in their inability to inspect packet payloads. Since the proxy operates within the kernel space, it is capable of efficiently scrutinizing packet headers in alignment with L3 to L4 policies. However, due to the inherent constraints within the kernel space, comprehensive inspection of the packet payload is not feasible. As a result, these solutions are required to redirect packets to additional components for payload inspection, contributing to substantial communication latency (see Section 5.1).

**User-level Proxy.** Istio [43] and Linkerd [44], initially designed for service mesh architectures [58], have been increasingly adapted to secure communications between containers. These solutions implement a user-level proxy, which literally operates within each container's net-NS in the user space. This proxy functions as a reverse proxy and offers extensive inspection capabilities for packet payloads, effectively restricting malicious L7 communications. In addition, these solutions can integrate with a kernel-level proxy. In this arrangement, the kernel-level proxy conducts the initial inspection of packet headers and then redirects the packets to the user-level proxy for a thorough payload examination.

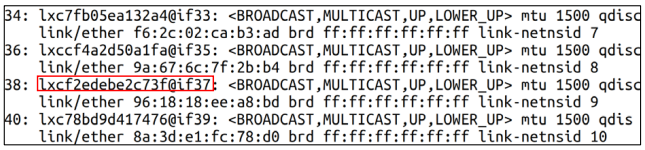
However, these solutions encounter difficulties when addressing attack vectors originating within the host net-NS (V2-3) due to the inherent limitations of their proxy implementations, which only operate within the container net-NS. Consequently, they lack visibility into packets that originate from the host net-NS. From a performance perspective, user-level proxies tend to operate at a slower pace compared to kernel-level proxies due to a variety of factors: the frequent context switching between kernel and user spaces, the overhead of packet copying, and the necessity for packets to traverse the entire network protocol stack twice — first in the kernel space and then again in the user space. Moreover, given that payload inspection is a resource-intensive task, user-level proxies can consume substantial system resources during the inspection, leading to substantial degradation in the available resources for microservices (see Section 5.1).

## 2.4 Motivating Example

In this section, we illustrate a network attack scenario to emphasize the severity of the attack vectors in today's containerized environments. The scenario, as depicted in Figure 2, involves two microservices (A and B) deployed in Kubernetes [30]. Each microservice consists of two real-world




**(a) Network interfaces visible to the normal container (Flask-A).**



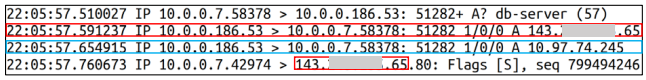

**(b) Network interfaces visible to the host-network container (HAProxy): The red box indicates the host-side interface associated with the victim (Flask-B).**

**Figure 3: Network interface visibility based on container types. A host-network container has unrestricted access to all interfaces within the host net-NS.**

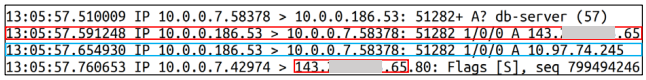
containers, Python Flask [5] and Redis [40]. In addition, Kubernetes deploys a host-network container, HAProxy, serving as an entry point to distribute incoming Internet traffic to the microservices. To enforce security policies restricting the communication across microservices, Cilium employs a kernel-level proxy, which operates within the host net-NS, filtering the outgoing traffic from each container.

**Attack Scenario.** Building upon previous studies [50, 63, 69], we take into account the presence of known vulnerabilities in containers. For instance, remote code execution vulnerabilities [6, 7] could be exploited to compromise Flask and HAProxy containers. We posit that an attacker has successfully gained access to the Flask-A and HAProxy containers and aim to execute a DNS spoofing attack targeting the victim, Flask-B. For this, the attacker uses both Flask-A (a normal container) and HAProxy (a host-network container). The attack process involves the attacker monitoring DNS requests made by the victim (Flask-B) to CoreDNS, which is the DNS server for Kubernetes. The attacker then crafts a spoofed response with a rogue server's IP address and injects this fabricated packet into the network directed towards the victim. The ultimate aim is to deceive the victim (Flask-B) into communicating with the rogue server, rather than the intended destination.

**From a Normal Container.** Despite the attacker's efforts to conduct a DNS spoofing attack from the compromised container (Flask-A), they are bound to encounter significant obstacles that render their attempts futile. First, the attacker can only access the container-side interface within their compromised container, meaning that it is impossible to monitor the victim's DNS requests (as illustrated in Figure 3a). Second, as depicted in Figure 2, the Cilium proxy would discard all attempts to send spoofed DNS responses to the victim container from the compromised one.




**(a) Packets captured from the victim's host-side interface (V2). The attacker can use the victim's host-side interface to monitor its DNS request and inject a spoofed DNS response.**




**(b) Packets captured from the container-side interface within Flask-B. The spoofed DNS response injected by the attacker is successfully received by the victim before the original DNS response arrives.**

**Figure 4: Results of DNS spoofing. The packets marked with red boxes are inserted by the attacker, and those with blue boxes are the packets sent by CoreDNS.**

**From a Host-network Container.** The situation changes drastically when the attacker is in a compromised host-network container (HAProxy) and conducts a DNS spoofing attack. As shown in Figure 3, HAProxy has extended visibility, offering insights into all host-side interfaces (V2) of the containers running on the host, including the software bridge (V3). This visibility enables the attacker to monitor traffic, including DNS requests, through the host-side interface of the victim container, regardless of net-NS isolation (refer to Figure 4a). Moreover, the attacker can directly inject spoofed DNS responses into the host-side interface of the victim. On the other hand, the kernel proxy deployed by Cilium cannot detect or filter these spoofed packets since the packets originate directly from the host net-NS rather than being transmitted from a specific container net-NS. As a result, the victim unknowingly accesses the rogue server set up by the attacker, as illustrated in Figure 4b.

**Summary.** This motivating example highlights the attack vectors within the host net-NS, which allows an attacker to compromise other containers on the same host without encountering any restrictions, despite the deployment of state-of-the-art security solutions.

# 3 *HELIOS* DESIGN

## 3.1 Assumptions and Threat Model

**Assumptions.** This work focuses on the possibility of attackers exploiting the breaches to conduct network-based attacks, enabling lateral movement from a compromised container to other containers, which is a significant concern as it grants attackers unauthorized control over other the compromised containers. Hence, we assume that the underlying infrastructure, including container platforms and the host operating system, is trustworthy. However, it is crucial to recognize that both normal and host-network containers can still pose

security risks. As numerous studies [47, 60, 63, 69] have already demonstrated the existence of known vulnerabilities in container images, including official images, available in public registries (e.g., Docker Hub), an attacker could exploit these vulnerabilities to compromise containers.

**Threat Model.** An attacker who has compromised a container can initiate network attacks on other benign containers through the container-side interface (V1). In a more severe case, if the attacker has compromised a host-network container, they can conduct extensive network attacks that target all containers within the same node by exploiting the attack vectors (V2-3) within the host net-NS. Note that compromising a host network container does not give the attacker root privileges. The primary security goal of our approach is to nullify all of these attack vectors conducted by both types of compromised containers and exert fine-grained control over network traffic between containers by enforcing stringent security policies. We exclude resource exhaustion attacks (e.g., DoS attacks) and system-driven attacks (e.g., container escape attacks) that exploit kernel vulnerabilities.

## 3.2 Design Requirements

Considering the limitations discussed in Section 2.3, we have pinpointed three requirements that *Helios* should meet.

**R1) Isolation of container communication channels**: The current container communication channels (e.g., a bridge network) are exposed to the host net-NS. This lack of isolation creates network attack vectors (V2-3) that an attacker, who has access to the host net-NS, can exploit. Therefore, a system should establish secure communication channels for containers, inherently isolated from the host net-NS, to eradicate the attack vectors effectively.

**R2) Granular network security enforcement**: Even with containers communicating over isolated channels, the risk remains that a compromised container could launch attacks on its neighbors within these channels. To counter this, a system should exercise granular control over the traffic flow across all network layers (from L3 to L7) within these isolated channels, all in alignment with given security policies.

**R3) High-performance and efficient security enforcement**: A system should deliver high-performance security enforcement for all container traffic within the host, without causing a performance drop for microservices. Furthermore, a system should conduct all the network security operations using minimal resources, thus preserving system resources for the reliable operation of microservices.

## 3.3 Design Overview

To fulfill the design requirements mentioned above, we propose *Helios*, a novel hardware-assisted security extension for container networks. Distinct from existing software-based solutions that rely on software proxies in the host system, *Helios* leverages the power of a smartNIC. This programmable network interface card, widely adopted in modern cloud environments, is typically used to offload network-centric tasks from the host CPU, thereby enhancing network performance. Going beyond this conventional usage, *Helios* leverages the smartNIC to establish physically isolated communication channels for containers and implement robust security enforcement within these channels, independent of the host system. As all container traffic is processed within the hardware itself (i.e., the smartNIC), *Helios* can effectively shield the container communication from potential interference by an attacker, no matter where the attacker is positioned within the container network. Moreover, this approach guarantees high-performance and efficient security enforcement without burdening the host's resources.

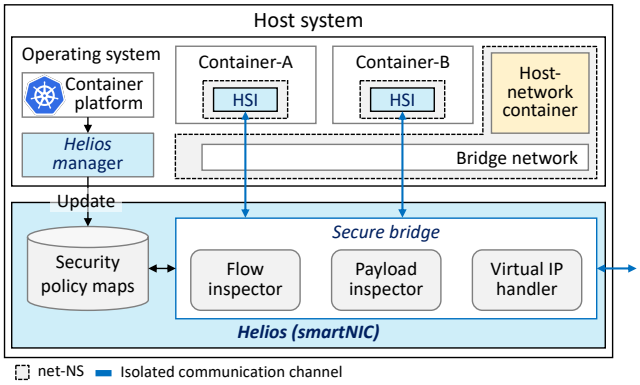



**Figure 5: The overall design of *Helios*. It consists of a *Helios* manager, a *Helios* secure bridge located on the smartNIC, and HSIs installed on each container.**

*Helios* comprises two main components: the manager and the secure bridge, as depicted in Figure 5. The manager is responsible for maintaining security policies given by an administrator. On the other hand, the secure bridge is implemented within the smartNIC and lays the foundation for the secure, isolated communication channel containers use. When a container requires secure communication, the manager automatically swaps out the container-side network interface for a specialized *Helios* secure interface, the *Helios* secure interface (HSI). This new interface enables direct packet exchanges between a container and the secure bridge inside the smartNIC, establishing an isolated communication channel that circumvents the host net-NS. This isolation effectively prevents traffic exposure to the host net-NS and satisfies the **R1**. In addition, the HSI authenticates the source of each packet transmitted by a container (**R3**) before forwarding it to the secure bridge, thus preventing the container from using unauthorized network addresses.

Within the secure bridge, incoming packets undergo thorough inspections, examining their packet headers and payloads. Any packets that violate given security policies (L3 to L7) are immediately discarded during the fine-grained policy control (**R2**). The secure bridge also translates virtual (private) network addresses within container packets to corresponding physical network addresses (e.g., encapsulation or NAT) using the virtual IP handler. This allows the filtered packets to be correctly routed to their intended destination. Once processed, these packets are dispatched from the secure bridge to the destination container's HSI or the external network. All these security enforcement operations are accelerated by dedicated hardware units integrated into the smartNIC, ensuring superior performance compared to existing software-based solutions while conserving system resources for microservices (**R3**).

## 3.4 *Helios* Manager

The *Helios* manager operates as the central coordinator for the secure bridge and *Helios* secure interfaces (HSIs) associated with containers by performing the following two roles.

*3.4.1 HSI Management.* One of the primary responsibilities of the *Helios* manager is to establish a secure connection between a container and the secure bridge. To accomplish this, the manager performs five steps, as shown in Figure 6. (1) Upon the creation of a new container, the container platform invokes the manager via the standard interface [17] and sends the new container's network information. (2) Subsequently, the manager installs a dedicated HSI with an inbuilt security engine into the new container's net-NS. (3) it then assigns the container's IP address to the newly installed HSI, ensuring the container can communicate seamlessly. (4) In parallel, the manager updates the security policy map in the secure bridge to reflect the security policies associated with the new container. (5) Finally, the manager disables both the container-side and host-side interfaces for the container, thereby effectively separating the container from the vulnerable bridge network, ensuring complete isolation from the host net-NS. When a container is terminated, the manager first removes the installed HSI and then the policies associated with the container stored in the secure bridge, ensuring the system configuration remains clean and updated.

Note that the *Helios* manager seamlessly integrates with the standard interface for network configuration of containers [17]. This integration allows the manager to configure all network settings for a new container prior to the initialization of containerized applications, preventing any network race conditions during the HSI installation process. Furthermore, the only modification made within a container during its connection to *Helios* is the replacement of the network interface. This approach requires no changes to the container

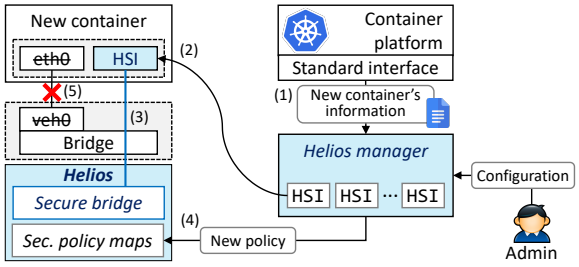



**Figure 6: An illustration of the HSI installation process.**

platform itself, upper-layer software stacks, and containerized applications, ensuring a transparent deployment process that does not disrupt the existing container ecosystem.

*3.4.2 Security Policy Management.* The manager also plays a crucial role in dynamically updating the security policy map within the secure bridge to reflect any changes in container configurations. To this end, it uses an event-driven approach to track the events associated with current security policies from the container platform. Specifically, the manager keeps an active watch on several types of events: (1) creation and termination of containers, (2) creation and removal of Kubernetes services [41], (3) addition and removal of hosts, and (4) application and removal of security policies. For each event detected, the manager generates a match-action table rule that the secure bridge can process. Through the PCI interface, the manager then pushes this newly created rule to the security policy map within the smartNIC.

The security policy map within the secure bridge comprises multiple sub-maps, each serving distinct stages in the secure bridge's pipeline. Consequently, each sub-map is updated and maintained independently within the hardware, facilitating an efficient policy update process. These maps are dynamically updated during runtime by the manager [55], and these updates can be executed without requiring the reprogramming of the secure bridge logic or the downtime of its packet processing pipeline. Note that during map updates, there exists the possibility that packets at a specific pipeline stage may be subject to the processing of previous map contents. However, this limitation has a marginal impact on the overall functionality of *Helios* and is a common challenge shared with all smartNIC-based solutions.

## 3.5 *Helios* Secure Interface

Container traffic is routed through the host-side interfaces and the software bridge within the host net-NS. In this design, packets sent from a container pass through the host net-NS, making them vulnerable to potential attacks by an attacker. To address this problem, *Helios* takes a novel approach by replacing the container-side interface with a specialized *Helios* secure interface (HSI), effectively isolating the container's

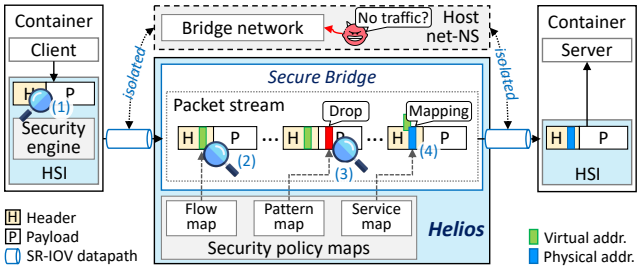



**Figure 7: The packet processing pipeline of *Helios*: (1) source verification, (2) L3/L4 inspection, (3) L7 inspection, and (4) virtual address mapping.**

communication from the host net-NS. Each HSI establishes a direct one-to-one link with the secure bridge, enabling direct packet exchange through a single root I/O virtualization (SR-IOV) datapath [1]. As shown in Figure 7, packets transmitted by an HSI are immediately forwarded to the RX buffer of the underlying smartNIC, bypassing the host net-NS and avoiding any need for processing within the host network stack. By implementing this isolated communication channel, *Helios* ensures that container traffic remains protected from an attacker who is in the host net-NS.

**Source Verification.** While the isolation of communication channels effectively protects container traffic, it is still possible that a malicious container connected to the secure bridge initiates attacks against another container. For this reason, a key role of the HSI is to prevent packet spoofing at the container level, a common first step in several network attacks. When the secure bridge receives a packet from an HSI, the context linked with the source container, except for the spoofable IP address, is removed. This context removal challenges the secure bridge in accurately preventing packet spoofing from containers connected to it. To overcome this hurdle, each HSI has an embedded security engine that checks the source address of outgoing packets. As shown by *step 1* in Figure 7, the engine inspects the packet sent from the HSI and verifies that the MAC and IP addresses match those assigned to the HSI. Any packets with invalid source addresses are discarded. In addition, this engine operates in the kernel space and conducts packet inspection before the network stack operations within the container net-NS occur. This design ensures high-performance inspection and guarantees the integrity of the source verification, even if the container using the HSI becomes compromised.

## 3.6 *Helios* Secure Bridge

The secure bridge, a core component of *Helios*, acts as the foundation for isolated container communication channels on a smartNIC. It also delivers fine-grained security controls on packets within the isolated channels by leveraging three specialized hardware units. Here, we delve into these hardware units within the secure bridge.

*3.6.1 Flow Inspector.* In microservice architectures, restricting the network visibility of containers is crucial for enhancing network security. The flow inspector is designed to fulfill this objective. The fundamental assumption of our security model is that containers within the same microservice are considered trustworthy to each other, while containers in different microservices might not be. Built on this assumption, the flow inspector isolates traffic between microservices to reduce the risk of lateral movement between microservices, thereby limiting the damage in case of a security breach.

The flow inspector executes a series of checks on incoming packets (step 2 in Figure 7). First, it verifies if the source and destination containers of the packet are part of the same microservice, using information from the security policy maps that detail active microservices. If not explicitly allowed by a specific policy, a packet sent from one microservice to another is discarded, ensuring traffic isolation at the microservice level. Then, the flow inspector inspects the packet headers, including protocols and ports, to verify that the connection between the source and destination containers is permitted. Packets violating the policies are discarded, blocking unauthorized network access at the container level.

*3.6.2 Payload Inspector.* Container communication often depends on L7 protocols such as HTTP REST APIs. Thus, it is essential to enforce security policies over L7 container communication. However, existing solutions face significant performance issues when enforcing L7 policies due to their reliance on software-based proxies. To tackle this, *Helios* introduces a payload inspector that enables fully offloaded enforcement of L7 policies for containers (step 3 in Figure 7).

The implementation of an efficient payload inspection mechanism on a smartNIC for enforcing L7 policies can be challenging due to the limited programmability and resource constraints of hardware architecture [54]. To overcome this challenge, we utilize a pre-compiled, state-machine-based pattern-matching approach, which is both resource-efficient and easily implementable on hardware. Our payload inspection mechanism, shown in Figure 8, consists of two main steps: (1) the pattern compilation step, which creates state machines based on given patterns, and (2) the pattern matching step, where the state machines are executed on the smartNIC to detect desired patterns in packet payloads.

As illustrated in the left part (A) of Figure 8, the pattern compilation step begins when the administrator applies new security policies. During this process, the manager extracts particular strings, which are required to match with payloads later, from given policies. These strings are then formatted into detection patterns ("GET /user/ HTTP 1.1..." as shown in Figure 8) with the help of a protocol adaptor, which accounts

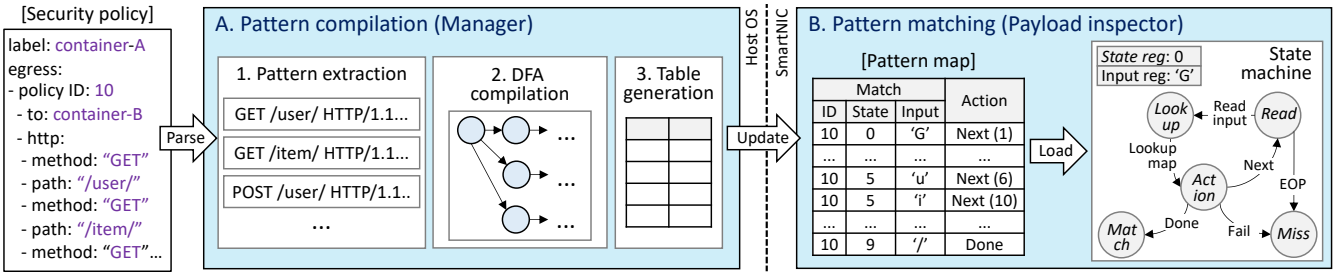



**Figure 8: The overall process of *Helios* secure bridge's payload inspection and *Helios* manager's pattern compilation.**

for the structure of the L7 protocol specified in the policies. Then, these detection patterns are compiled into a deterministic finite automaton (DFA) that shares their common prefixes. The state transition table of the DFA is converted into a match-action table and then inserted into the pattern map of the secure bridge.

When the secure bridge receives a packet subject to L7 policies, the payload inspector uses a state machine to execute the compiled DFAs. As displayed in the right part (B) of Figure 8, the state machine consists of five internal states and two registers. In the Read state, the state machine reads a single byte from the input pointer and stores it in the input register, subsequently transitioning to the Lookup state. If the input pointer hits the maximum length or reaches the end of the packet (EOP), the state machine moves to the Miss state. During the Lookup state, the action tied to the current input and state registers is retrieved from the pattern map, leading to the Action state. If the lookup fails, the state machine moves to the Miss state. If the retrieved action is labeled *Next*, the state register updates, and the state machine cycles back to the Read state. If the fetched action is *Done*, the machine moves to the Match state. Finally, the state machine ends either in the Match or Miss state, signifying the detection or absence of a pattern in the payload. After this, the packet is handled (either drop or pass) in accordance with the specified action in the L7 policy.

*3.6.3 Virtual IP Handler.* Container platforms often use a service IP address [41], which associates a steady virtual IP address with a group of containers (referred to as backends). This service IP address stays the same even when the containers are replaced, providing a consistent endpoint for accessing the applications that run on these containers. The virtual IP handler within our system plays the key role of mapping packets associated with service IP addresses to the corresponding backend containers, ensuring the packets are properly processed (step 4 in Figure 7). The virtual IP handler initially identifies packets linked with service IP addresses by referencing the information about active services stored in the service sub-map (as illustrated at the bottom of Figure 7). Once identified, it performs Destination Network Address Translation (DNAT) by rewriting the destination IP address and port of the packet to match the actual IP address and service port of the corresponding backend container. The selection of the appropriate backend container is based on the hash value of the packet header, ensuring consistent network sessions between containers. In addition to this, the connection information for the packet is stored within the secure bridge, allowing for efficient handling of packets moving in the reverse direction (through SNAT).

The virtual IP handler also plays a crucial role in encapsulating packets that are intended for containers on different hosts using tunneling protocols such as VXLAN. This encapsulation is necessary because the intermediate network switches do not recognize the private addresses of containers. These addresses need to be packaged within an additional header that includes the IP address of the physical host. When a packet is destined for a container on a different host, the virtual IP handler adds an outer header to the original header of the packet. This outer header includes the IP addresses of the source and destination hosts. The handler uses the host address sub-map, located within the security policy map, to perform this encapsulation. This encapsulation process allows the packet to travel through physical switches and eventually reach its intended destination. When the packet arrives at the receiving end, the outer header is removed. The container then receives the original packet, without the additional encapsulated information.

## 4 SECURITY EVALUATION

In this section, we aim to demonstrate the security effectiveness of *Helios* by addressing the following key questions:

- Q1: Can the isolated container communication channel offered by *Helios* effectively prevent container traffic exposure and malicious traffic injection from the host net-NS?
- Q2: Does the source verification mechanism implemented in the HSI ensure the identity of each packet and effectively block packet spoofing attacks originating from containers?

- Q3: Can the secure bridge limit the network visibility of individual containers and effectively mitigate network attacks across various protocols ranging from L3 to L7?

## 4.1 Prototype Implementation

We have implemented a full prototype of *Helios* to evaluate its feasibility and effectiveness. The secure bridge has been realized using the Netronome Agilio CX 2x40GbE NIC [36], a commodity ASIC-based smartNIC. We have developed three hardware units within the secure bridge with 1.5K lines of XDP-eBPF code offloaded to the smartNIC using eBPF offloading technology [55]. The security policy map is implemented using four eBPF maps: (1) flow map, (2) pattern map, (3) service map, and (4) host address map. These eBPF maps are also offloaded onto the smartNIC and updated in run time by the *Helios* manager via the eBPF subsystem [55]. The manager is implemented with 2K lines of Python code and operates atop of Container Network Interface (CNI) [17] for network configuration of containers running on its node. During the HSI installation process, the manager deploys an SR-IOV virtual function (VF) with an eBPF program to a newly created container. The eBPF program is attached to the traffic control layer (TC) [67] of the VF and performs source verification. The manager employs Kubernetes APIs [31] to identify container events associated with the current policies.

Victim's view (Flask-B)

```
root@flask-b:/home# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
37: eth0@if38: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue st
```

Attacker's view (HAProxy)

```
root@haproxy:/home/attacker# ip link | grep 38:
38: lxcf2edebe2c73f@if37: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc

root@haproxy:/home/attacker# tcpdump -i lxcf2edebe2c73f tcp
…
04:01:26.975255 IP host-A.24363 > 10.0.0.155.http: Flags [S], seq 0, win 81
04:01:26.975303 IP 10.0.0.155.http > host-A.24363: Flags [S.], seq 42441029
…
root@haproxy:/home/attacker# ./scan.py -t 10.0.0.155 -i lxcf2edebe2c73f
80 | Open, 8080 | Closed, 443 | Closed
```

**(a) The attacker can exploit the host-side interface (the red box) of the victim to inject scanning packets and monitor the activated ports.**

Victim's view (Flask-B)

```
root@flask-b:/home# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
42: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
```

Attacker's view (HAProxy)

```
root@haproxy:/home/attacker# ip link | grep 38:
root@haproxy:/home/attacker#  <-- No host-side interface for Flask-B

root@nss-All-Series:/home/nss# tcpdump -i lxcf2edebe2c73f tcp
tcpdump: lxcf2edebe2c73f: No such device exists
root@haproxy:/home/attacker# ./scan.py -t 10.0.0.155 -i lxcf2edebe2c73f
No route to 10.0.0.155
```

**(b) Once the HSI is installed, the host-side interface is eliminated, effectively thwarting the attacker's subsequent attacks.**

**Figure 9: Effectiveness of the isolation of container communication channel provided by *Helios*.**

## 4.2 Isolation of Communication Channels

*Helios* ensures the physical isolation of inter-container communication from the host net-NS. To address the first question (**Q1**), we evaluate the effectiveness of our system's isolated communication channel. In this evaluation, we consider the same scenario of our motivation (Section 2.4) where the host network container (HAProxy) is compromised by the attacker. Figure 9a illustrates the exposure of the victim's host-side interface (the interface with ID 38) in the host net-NS, which presents an opportunity for the attacker with HAProxy to launch various attacks on the victim. The attacker can initiate port scanning attacks to identify the active ports of the victim by injecting probe packets into the host-side interface and monitoring the response traffic. This enables the attacker to gather network information about the victim container, facilitating the launch of subsequent network attacks. Notably, existing solutions lack visibility into the injected probe packets and are unable to restrict the attacker from using the host-side interface to eavesdrop on the victim's packets, as discussed in Section 2.3.

However, with the installation of the HSI on the victim container within our system, its network communication becomes isolated from the host net-NS, where the attacker is situated. As depicted in Figure 9b, the victim's HSI has no host-side interfaces and instead directly connects to the secure bridge. Consequently, the attacker cannot eavesdrop on or interfere with the victim's traffic, resulting in the failure of further network attacks targeting the victim.

## 4.3 Packet Source Verification

To address **Q2**, we assess the effectiveness of source verification in HSIs against packet spoofing attacks. In this evaluation, we deploy HSIs on each container in the scenario illustrated in Figure 2. We assume that Flask-A, which is compromised by an attacker, attempts to conduct network attacks targeting other containers connected to the secure bridge. As depicted in Figure 10a, Flask-A (the attacker) impersonates Flask-B (10.0.0.155) and sends malicious TCP RST packets to the victim (Redis-B) in a different microservice, utilizing packet spoofing. Without the source verification, the secure bridge lacks visibility into the source container of these spoofed packets and thus cannot discard them. Consequently, the victim receives the spoofed packets, leading to the termination of active TCP sessions. However, when source verification is enabled, the HSI installed on Flask-A rejects all outgoing packets with spoofed addresses different from its assigned address (10.0.0.142), as shown in Figure 10b.

The attacker (Flask-A) may attempt to bypass the source verification by disabling the security engine of the HSI. However, the security engine is designed to ensure its operation

Outgoing packets of the *attacker* container (Flask-A)

```
02:39:19.699333 IP 10.0.0.155.39444 > 10.0.0.163.6379: Flags [R.], seq 1
02:39:19.699338 IP 10.0.0.155.39444 > 10.0.0.163.6379: Flags [R.], seq 1
```

Packets received from the *victim* container (Redis-B)

```
02:39:19.699226 IP 10.0.0.163.6379 > 10.0.0.155.39444: Flags [.], ack 1,
02:39:19.699295 IP 10.0.0.155.39444 > 10.0.0.163.6379: Flags [P.], seq 1:
02:39:19.699339 IP 10.0.0.155.39444 > 10.0.0.163.6379: Flags [R.], seq 1
02:39:19.699343 IP 10.0.0.155.39444 > 10.0.0.163.6379: Flags [R.], seq 1
```

**(a) The spoofed TCP RST packets sent by the attacker (red boxes) are received from the victim, terminating active TCP sessions.**

Outgoing packets of the *attacker* container (Flask-A)

```
02:76:26.680786 IP 10.0.0.155.28521 > 10.0.0.163.6379: Flags [R.], seq 1
02:76:26.680789 IP 10.0.0.155.28521 > 10.0.0.163.6379: Flags [R.], seq 1
```

Packets received from the *victim* container (Redis-B)

```
02:76:26.680791 IP 10.0.0.155.28521 > 10.0.0.163.6379: Flags [P.], seq 1:
02:76:26.680838 IP 10.0.0.163.6379 > 10.0.0.155.28521: Flags [.], ack 75,
02:76:26.680915 IP 10.0.0.163.6379 > 10.0.0.155.28521: Flags [P.], seq 1:
02:76:26.680947 IP 10.0.0.155.28521 > 10.0.0.163.6379: Flags [.], ack 72
```

**(b) The HSI drops the spoofed TCP RST packets during the source verification. Thus, these packets are not delivered to the victim.**

```
root@host-a:/home# tc filter show dev flask_b_hsi egress
filter protocol all pref 49152 bpf chain 0 handle 0x1 hsi.o:[source
_verification] direct-action not_in_hw id 17 tag 3b185187f1855c4c
---------------------------------------------------------------------------

root@flask-a:/home# tc filter del dev eth0 egress
RTNETLINK answers: Operation not permitted
```

Unable to disable the security engine of the HSI

**(c) The security engine (*hsi.o*) of the HSI ensures the integrity of the source verification, even if the container (Flask-a) is compromised.**

**Figure 10: Effectiveness of source verification In a HSI.**

```
root@flask-a:/attack_tool# ./container-scanning -n 10.0.0.0/24
ARP scanning results: 4 host alive
10.0.0.117      7a:58:60:aa:cb:81       10.0.0.129      96:18:18:ee:a8:bd
10.0.0.155      3c:fd:fe:c4:93:39       10.0.0.163      9c:5c:8e:4f:09:61

ICMP scanning results: 4 host alive
10.0.0.117      10.0.0.129      10.0.0.155      10.0.0.163
```

Microservice-B

**(a) Without flow inspector, the attacker (Flask-A) can discover and initiate communication with containers within another microservice.**

```
root@flask-a:/attack_tool# ./container-scanning -n 10.0.0.0/24
ARP scanning results: 2 host alive
10.0.0.117      7a:58:60:aa:cb:81       10.0.0.129      96:18:18:ee:a8:bd

ICMP scanning results: 2 host alive
10.0.0.117      10.0.0.129
```

**(b) The flow inspector effectively prevents the attacker from discover irrelevant containers within another microservice.**

**Figure 11: Effectiveness of the flow inspector inside the secure bridge against network scanning attacks.**

even within a compromised container. As depicted in the upper part of Figure 10c, the *Helios* manager attaches the security engine, implemented as an eBPF program, to the TC hook of the HSI prior to its installation into the container. Disabling the eBPF program requires system root privileges. Consequently, the attacker is unable to disable the security engine attached to the HSI within Flask-A, as shown in the bottom part of Figure 10c.

```
05:26:01.329141 IP 10.0.0.117.33266 > 10.0.0.155.80: Flags [.], ack 1
05:26:01.331563 IP 10.0.0.117.33266 > 10.0.0.155.80: Flags [P.], seq 1:7
length 80: HTTP: DELETE /user/1 HTTP/1.1
05:26:01.331912 IP 10.0.0.155.80 > 10.0.0.117.36818: Flags [P.], seq 1:3
length 83: HTTP: HTTP/1.1 200 OK
```

Unallowed API request

Response for the request

**(a) Without the payload inspector, unallowed API requests from the attacker (Flask-A) are successfully delivered to the victim.**

```
11:15:15.432100 IP 10.0.0.117.33266 > 10.0.0.155.80: Flags [P.], seq 1:7
length 83: HTTP: DELETE /user/1 HTTP/1.1
11:15:15.712106 IP 10.0.0.117.33266 > 10.0.0.155.80: Flags [P.], seq 1:7
length 83: HTTP: DELETE /user/1 HTTP/1.1
11:15:15.958245 IP 10.0.0.117.33266 > 10.0.0.155.80: Flags [P.], seq 1:7
length 80: HTTP: GET /user/1 HTTP/1.1
11:15:15.969124 IP 10.0.0.155.80 > 10.0.0.163.36818: Flags [P.], seq 1:3
length 145: HTTP: HTTP/1.1 200 OK
```

Unallowed API requests without responses

Allowed API request and response

**(b) The payload inspector effectively blocks the unallowed API requests while allowing other benign requests.**

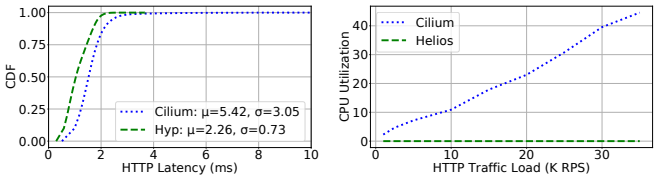
**Figure 12: Effectiveness of the payload inspector's L7 security policy enforcement (the attacker's view).**

## 4.4 Security Policy Enforcement

Here, we address **Q3** by demonstrating the effectiveness of the secure bridge's security policy enforcement.

*4.4.1 Network Layer.* The flow inspector within the secure bridge restricts the network visibility and reachability of containers. Without the flow inspector, as depicted in Figure 11a, a malicious container (Flask-A) can conduct network scanning attacks to discover containers in another microservice that are not allowed to communicate. This unrestricted network visibility can facilitate further attacks against the discovered containers. However, with the flow inspector enabled, the attacker's network visibility is confined to microservice-A, and any network connections initiated by the attacker are controlled based on predefined policies. As shown in Figure 11b, the attacker can only discover and communicate with containers within the same microservice.

*4.4.2 Application Layer.* While the flow inspector focuses on the network layer, the payload inspector provides application-layer security by examining packet payloads, preventing unauthorized access over L7 protocols. To demonstrate the effectiveness of the payload inspector, we assume that all four containers in the scenario can communicate with each other, and Flask-B offers two HTTP-based APIs: {Method: GET, URI: /user} and {Method: DELETE, URI: /user}. Here, the user deletion API is allowed for microservice-B containers and cannot be invoked by any other container. Without the payload inspector, Flask-A can invoke the unallowed user deletion API, as shown in Figure 12a. However, with the payload inspector enabled and L7 policies configured to restrict access to the user deletion API, all HTTP traffic toward Flask-B is subjected to the scrutiny of the payload inspector. It inspects the payloads of incoming HTTP traffic and discards request packets for the user deletion API sent from




**(a) Inter-container HTTP latency.** **(b) CPU utilization.**

**Figure 13: End-to-end security enforcement performance of *Helios* and Cilium.**

**Table 2: HTTP throughput measurements with *Helios*'s payload inspector and Cilium's user-level proxy.**

| *Throughput (RPS)* | *Cilium* | *Cilium (L7)* | *Helios* | *Helios* (L7) |
|---|---|---|---|---|
| Intra (single)-host | 54.8K | 27.4K (-50%) | 79.6K | 78.4K (-1.5%) |
| Inter-host | 51.2K | 25.8K (-51%) | 74.5K | 73.7K (-1.5%) |

unauthorized containers, while still allowing other benign packets (e.g., packets for the user search API), as depicted in Figure 12b. As a result, the payload inspector effectively controls network access between containers over L7 protocols.

## 5 PERFORMANCE EVALUATION

In this evaluation, we assess the performance advantages of *Helios* by conducting comparisons with state-of-the-art solutions. We begin by measuring the end-to-end security enforcement performance of our system. Subsequently, we conduct a comprehensive performance analysis using microbenchmarks. For the evaluation, we establish a Kubernetes environment with Cilium CNI setting [15] on two machines equipped with an Intel Xeon E5-2630v4 CPU and 64 GB of RAM. Each machine is equipped with a Netronome 40 Gbps NIC [36], and they are connected via these NICs.

### 5.1 E2E Security Enforcement Performance

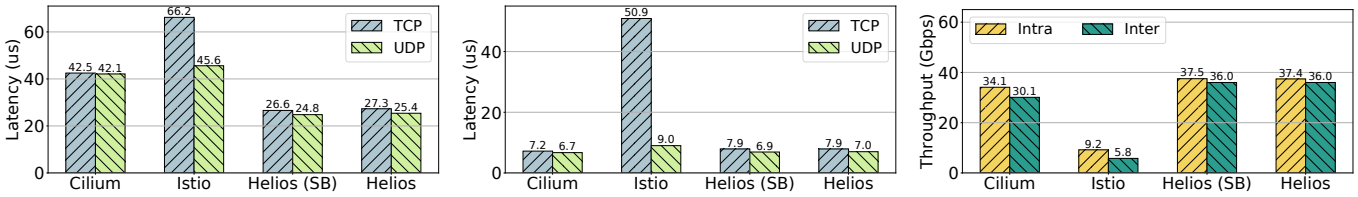
We conduct a performance comparison between *Helios* and Cilium, which is known for its high performance among modern solutions. The experiment involves running web server [37] and client containers on the same host, with a security policy in place to restrict communication between them to specific ports (80) and URLs (*GET: test.html*). We measure the HTTP latency and throughput of both Cilium and *Helios* using the wrk2 [3] tool.

Figure 13a presents the CDF of HTTP latency observed during the experiment at a load of 10K requests per second (RPS). Cilium exhibits a latency ranging from 1.1 *ms* to 10.3 *ms*, with a mean of 5.42 *ms* and a standard deviation of 3.05 *ms*. Notably, approximately 20% of packets experience latency above 2.0 *ms*, and 1% of packets encounter latency exceeding 3.46 *ms*, indicating the presence of long tail latency. This poor performance is attributed to the software-based L7 security enforcement employed by Cilium's user-level proxy [14]. In contrast, *Helios* demonstrates an average HTTP latency that is two times faster and exhibits a more consistent distribution of latency. Moreover, approximately 90% of packets are processed within 1.7 *ms*, with only 1% of packets exceeding 2.0 *ms*. This improvement can be attributed to *Helios*'s hardware-based payload inspection capability. Next, we conduct throughput measurements comparing the amount of HTTP traffic that both systems can inspect. As shown in Table 2, Cilium's HTTP throughput notably decreases by 50.0% when the L7 security policy is enabled. This performance degradation is consistent with the observed latency results mentioned earlier. The software-based nature of Cilium's user-level proxy introduces overhead and affects the throughput. In contrast, *Helios* shows a slight decrease in performance of only 1.5% when the payload inspector is enabled. Moreover, *Helios* exhibits the capability to inspect 2.9 times more HTTP request packets compared to Cilium.

Lastly, we conduct measurements to assess the CPU utilization of *Helios* and Cilium as the HTTP traffic sending rate increases. We employ the mpstate tool [34] to monitor the CPU resources utilized by the user-level proxy of Cilium during the experiment. As shown in Figure 13b, the CPU utilization of Cilium shows a linear growth pattern, reaching 44% of the total host CPU capacity when subjected to a 35 Mbps HTTP traffic rate. This observation implies that existing solutions, such as Cilium, can consume a significant amount of resources for only network security checks, exceeding 40% of the total CPU capacity. The substantial CPU utilization can impede efficient resource allocation, consequently degrading the overall performance of microservices. Also, this correlation between the increase in HTTP delay, the drop in throughput observed in Cilium, and its high CPU consumption are evident from our findings. In contrast, *Helios* leverages the smartNIC to inspect packet payloads, preserving host CPU resources for microservice operation.

### 5.2 Microbenchmark

In Helios, container traffic undergoes the smartNIC to establish a physically isolated communication channel. This design may introduce a limitation where both inter- and intra-host container traffic should be routed through the smartNIC. Consequently, this additional routing process can increase latency in the intra-host container communication scenario. To assess the impact of this overhead, we conduct comparison experiments by disabling all security measures of *Helios* and other solutions (a simple forwarding mode).




**(a) Inter-host communication latency.** **(b) Intra-host communication latency.** **(c) TCP communication throughput.**

**Figure 14: Latency and throughput measurements in inter-host and intra-host container communication scenarios, with security measures disabled. Helios (SB): secure bridge only, Helios: secure bridge and source verification.**

This allowed us to focus solely on the traffic forwarding performance of the system in both intra- and inter-host container communication scenarios. We use netperf [27] and iperf [29] for latency and TCP throughput measurements.

**Inter-host Communication.** In inter-host container communications, *Helios* leverages its hardware-based design to outperform the existing solutions in terms of network throughput and latency. Figure 14a demonstrates that *Helios* achieves about 1.5 times lower TCP latency and about 1.65 times lower UDP latency compared to Cilium. Furthermore, compared to Istio, *Helios* exhibits network latency approximately 6.4 times faster for TCP packets and about 1.71 times faster for UDP packets. The superior performance of *Helios* in inter-host container communication can be attributed to its hardware-based packet encapsulation and transmission approach. When packets are delivered to another host, Cilium and Istio require traversal through multiple network stacks. This introduces extra packet processing overhead, including packet encapsulation, at the host level. In contrast, *Helios* encapsulates packets and transmits them to another node at the hardware level, utilizing the capabilities of the smartNIC to maximize performance. Also, as shown in Figure 14c, the throughput achieved with *Helios* is close to the maximum bandwidth (40 Gbps). In comparison, Cilium and Istio show throughputs of only 30.2 Gbps and 5.8 Gbps, respectively.

**Intra-host Communication.** Figure 14b illustrates the round-trip times observed during intra-host communication between containers. The results reveal that *Helios* introduces approximately 10% more latency in TCP communication and around 5% more latency in UDP communication compared to Cilium. This overhead is attributed to the additional packet copying operations and PCI traversal required by *Helios* to forward container traffic to the secure bridge located on the smartNIC. In contrast, Cilium directly forwards container traffic at the network interface level, minimizing packet forwarding overhead. However, in comparison with Istio, which employs user-level proxies, *Helios* exhibits significantly lower latency, achieving six times faster performance in TCP communication and approximately 12% faster performance in UDP communication. This advantage stems from the substantial overhead incurred by Istio in redirecting all TCP traffic from containers to the user-level proxy. Similarly, as demonstrated in Figure 14c, *Helios* outperforms Cilium by about 10% in terms of TCP throughput and surpasses Istio by four times in the same metric. This performance benefit is due to *Helios* leveraging dedicated hardware resources for traffic processing. In conclusion, *Helios* incurs a performance degradation of up to 10% in intra-host communications. However, this penalty is negligible considering the robust security provided by *Helios* and the performance benefits derived from offloading security measures.

**Source Verification Overhead.** Finally, we assess the overhead introduced by the source verification feature of the HSI. We compare the network performance of *Helios* without source verification to a fully functional *Helios*. We observe that this feature incurs negligible overhead in both intra-host and inter-host communication scenarios. Specifically, the overhead ranges from 0.2 *ms* (intra-host) to 4.6 *ms* (inter-host) as shown in Figure 14a and 14b, respectively. This negligible overhead introduced is well within acceptable limits, considering the enhanced security benefits it provides.

**HSI Installation Overhead.** The *Helios* manager installs an HSI to a new container to connect it to our system. The overhead that occurs during this HSI installation process should also not affect the agile deployment of containers. To evaluate the impact of this overhead, we measure the average times for the HSI installation process during 100 container creations. The installation process is divided into two phases: the HSI deployment phase and the associated policy update phase. Based on our measurements, the total HSI installation process exhibits an average time of 366ms. Specifically, the HSI deployment phase consumes 349ms (95.3% of the total), while the policy update phase necessitates 17ms (4.7% of the total). It is noteworthy that even the startup of a simple container (i.e., the "Hello World" container) in Kubernetes takes several seconds [4]. In this context, the overhead introduced by the HSI installation process can be considered negligible.

# 6 RELATED WORK

**Container Network Security.** Recently, several studies [48, 59, 64, 65] have examined the network security aspects of container environments and have highlighted the vulnerabilities in current container networks. To address these threats, various approaches have been proposed. Nam et al. introduced Bastion [65], which utilizes per-container security proxies implemented with recent kernel networking features, such as XDP-eBPF, to provide low-overhead L3/L4 security inspection for containers. Another line of research [49, 75] has focused on providing more granular network policy enforcement by considering application-level context, such as the version of the SSL library used by containers. However, as discussed in Section 2.3, these studies have no security measures for network attacks originating from the host net-NS. In addition, most of them suffer from significant performance degradation due to the limited L7 inspection capability of their software-based proxies. In contrast to these studies, our work adopts a hardware-based approach, effectively safeguarding containers from network attacks originating from the host net-NS, while enhancing L7 inspection performance by leveraging the processing power of a smartNIC.

**Hardware-assisted Network Security.** With recent advancements in programmable data plane technology, researchers have been exploring the offloading of network security functions to the data plane in cloud environments [46, 53, 66, 72, 74]. These studies leverage programmable switches or smartNICs to provide various security functions to hosts and tenant applications on it. For example, P4DDPI [46] implements DNS traffic analysis logic on P4 switches to offer high-performance inspection of DNS packets and protect connected hosts from DNS-related attacks. However, most of these studies are tailored for virtual machine-based environments and are not directly applicable to container environments. In container environments, where multiple containers share the same kernel and communicate through a virtualized network, these solutions face challenges. For instance, studies using P4 switches [46, 72] lack visibility into intra-host container communication, limiting their ability to prevent network attacks between containers on the same host. Likewise, smartNIC-based studies [53, 66, 74] do not have direct packet exchange capabilities with containers and lack visibility into container communication contexts, such as service IPs [41]. In contrast, *Helios* is designed to address the unique challenges of container environments.

# 7 DISCUSSION AND LIMITATIONS

**Scalability.** One limitation of *Helios* is the amount of memory available in the underlying smartNIC. The number of policies and active L4 flows that can be maintained at line-rate is limited by the smartNIC's memory. In particular, *Helios* can track around 1M L4 flows and enforce a maximum of 8K policies per host. For a larger number of policies, external memory access is necessary, which may incur overhead during policy lookup. The number of containers or groups of containers sharing the same net-NS (i.e., pods) that can be connected to *Helios* is also limited by the number of SR-IOV VFs supported by the smartNIC. However, modern smartNICs typically offer up to 255 VFs per port. Given that Kubernetes supports up to 110 pods per host [32], *Helios*, when deployed on a host, can support all the containers running on that host without encountering scalability issues.

**PCIe overhead.** In *Helios*, both inter- and intra-host container traffic are processed by the smartNIC. A drawback of this design is that even intra-host container traffic should be passed to the smartNIC via the PCIe data path, potentially incurring PCIe bandwidth consumption and extra latency in intra-host communication. However, as demonstrated in Section 5.2, the extra latency overhead introduced by *Helios* is just a few microseconds. In light of the robust security measures afforded by *Helios* and the performance advantages derived from offloading CPU-intensive security tasks, this latency overhead can be deemed negligible. Regarding PCIe bandwidth consumption, modern smartNICs typically support PCIe version 4 x16 lanes, affording a PCIe bandwidth of 250Gb/s. This capacity ensures that *Helios* allocates 100 Gb/s for high-performance inter-host communication while still retaining significant PCIe bandwidth (about 150Gb/s) for managing intra-host communication.

# 8 CONCLUSION

In modern cloud environments, containerization is prevalent. However, existing network security solutions for containers grapple with performance constraints and emerging attack vectors. To address this, we introduce *Helios*, a novel hardware-assisted network security extension tailored for containers. It ensures the physical isolation of container communications from the host system using a hardware-based network bridge with embedded security inspectors. Comprehensive evaluations demonstrate *Helios*'s effectiveness against diverse network attacks and its performance superiority over existing solutions, marking a significant advancement in container network security.

# 9 ACKNOWLEDGEMENT

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2023-RS-2023-00258649) and the ICAN (ICT Challenge and Advanced Network of HRD) support program (IITP-2023-RS-2023-00259867) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).

# REFERENCES


[1] 2008. PCI-SIG Single Root I/O Virtualization (SR-IOV) Support in Intel® Virtualization Technology for Connectivity. https://www.intel.com/content/www/us/en/pci-express/pci-sig-single-root-io-virtualization-support-in-virtualization-technology-for-connectivity-paper.html.

[2] 2013. Namespaces in Operation, Part 1: Namespaces Overview. https://lwn.net/Articles/531114/.

[3] 2014. wrk – a HTTP Benchmarking Tool. https://github.com/wg/wrk.

[4] 2015. Kubernetes Performance Measurements and Roadmap. https://kubernetes.io/blog/2015/09/kubernetes-performance-measurements-and/.

[5] 2018. Flask Docker Container Image. https://hub.docker.com/r/jcdemo/flaskapp.

[6] 2019. CVE-2019-8341. https://nvd.nist.gov/vuln/detail/CVE-2019-8341/.

[7] 2020. CVE-2020-11100. https://nvd.nist.gov/vuln/detail/CVE-2020-11100/.

[8] 2021. State of Kubernetes Security Report. https://thechief.io/c/editorial/state-of-kubernetes-security-report.

[9] 2022. 7 Most Infamous Cloud Security Breaches. https://blog.storagecraft.com/7-infamous-cloud-security-breaches/.

[10] 2022. Amazon Web Services. https://aws.amazon.com/.

[11] 2023. AppArmor, Linux Kernel Security Module. https://apparmor.net/.

[12] 2023. bridge(8) — Linux manual page. https://man7.org/linux/man-pages/man8/bridge.8.html.

[13] 2023. Calio-felix. https://docs.projectcalico.org/reference/felix/.

[14] 2023. Cilim Envoy Extension. https://docs.cilium.io/en/v1.13/security/network/proxy/envoy/.

[15] 2023. Cilium. https://www.cilium.io/.

[16] 2023. Cilium-agent. https://docs.cilium.io/en/stable/cmdref/cilium-agent/.

[17] 2023. CNI: The container network interface. https://www.cni.dev/.

[18] 2023. Docker. https://www.docker.com.

[19] 2023. Docker host networking. https://docs.docker.com/network/host/.

[20] 2023. DockerHub: envoyproxy/envoy. https://hub.docker.com/r/envoyproxy/envoy.

[21] 2023. DockerHub: hashicorp/boundary. https://hub.docker.com/r/hashicorp/boundary.

[22] 2023. DockerHub: sysdig. https://hub.docker.com/r/sysdig/sysdig.

[23] 2023. eBPF Introduction, Tutorials. https://docs.cilium.io/en/stable/bpf/.

[24] 2023. Flannel-d. https://github.com/flannel-io/flannel.

[25] 2023. Google Cloud Platform (GCP). https://cloud.google.com/.

[26] 2023. HAProxy ingress controler. https://haproxy-ingress.github.io/.

[27] 2023. Hewlett Packard Enterprise. Netperf. https://hewlettpackard.github.io/netperf/.

[28] 2023. Host network driver. https://docs.docker.com/network/drivers/host/.

[29] 2023. iPerf. Network Bandwidth Measurement Tool. https://iperf.fr/iperf-download.php.

[30] 2023. Kubernetes. https://kubernetes.io.

[31] 2023. Kubernetes API Watcher Design. https://docs.openstack.org/kuryr/0.2.0/devref/k8s_api_watcher_design.html.

[32] 2023. Kubernetes: Considerations for large clusters. https://kubernetes.io/docs/setup/best-practices/cluster-large/.

[33] 2023. Kubernetes Privilege Escalation. https://i.blackhat.com/USA-22/Thursday/US-22-Avrahami-Kubernetes-Privilege-Escalation-Container-Escape-Cluster-Admin.pdf.

[34] 2023. Linux SYSSTAT. http://sebastien.godard.pagesperso-orange.fr/.

[35] 2023. Microsoft Azure. https://azure.microsoft.com/.

[36] 2023. Netronome Agilo CX smartNIC 2x40GbE. https://www.netronome.com/media/documents/PB_NFP-4000-7-20.pdf.

[37] 2023. Nginx Docker Container. https://hub.docker.com/_/nginx.

[38] 2023. OpenVPN Access Server. https://hub.docker.com/r/mace/openvpn-as.

[39] 2023. Project Calico. https://www.projectcalico.org/.

[40] 2023. Redis Docker Container. https://hub.docker.com/_/redis.

[41] 2023. Service | Kubernetes. https://kubernetes.io/docs/concepts/services-networking/service/.

[42] 2023. TCPdump manpage. https://www.tcpdump.org/manpages/.

[43] 2023. The Istio service mesh. https://istio.io/.

[44] 2023. The Linked servie mesh. https://linkerd.io/.

[45] 2023. veth – Virtual Ethernet Device. https://man7.org/linux/man-pages/man4/veth.4.html/.

[46] Ali AlSabeh, Elie Kfoury, Jorge Crichigno, and Elias Bou-Harb. 2022. P4DDPI: Securing P4-Programmable Data Plane Networks via DNS Deep Packet Inspection. In *Proceedings of the 2022 Network and Distributed System Security (NDSS) Symposium. 1ŝ7.*

[47] Kelly Brady, Seung Moon, Tuan Nguyen, and Joel Coffman. 2020. Docker Container Security in Cloud Computing. In *In Proceedings of Annual Computing and Communication Workshop and Conference.* 975–980.

[48] Gerald Budigiri, Christoph Baumann, Jan Tobias Mühlberg, Eddy Truyen, and Wouter Joosen. 2021. Network Policies in Kubernetes: Performance Evaluation and Security Analysis. In *In proceedings of Joint European Conference on Networks and Communications & 6G Summit.* 407–412.

[49] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. 2020. Valve: Securing Function Workflows on Serverless Computing Platforms. In *Proceedings of the Web Conference 2020.* 939–950.

[50] Ana Duarte and Nuno Antunes. 2018. An Empirical Study of Docker Vulnerabilities and of Static Code Analysis Applicability. In *In Proceedings of Latin-American Symposium on Dependable Computing.* 27–36.

[51] William Findlay, David Barrera, and Anil Somayaji. 2021. BPFContain: Fixing the Soft Underbelly of Container Security. *arXiv preprint arXiv:2102.06972* (2021).

[52] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *Proceedings of International Symposium on Research in Attacks, Intrusions and Defenses.* 443–458.

[53] Joel Hypolite, John Sonchack, Shlomo Hershkop, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. 2020. DeepMatch: practical deep packet inspection in the data plane using network processors. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies.* 336–350.

[54] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. 2019. Fast string searching on pisa. In *Proceedings of ACM Symposium on SDN Research.* 21–28.

[55] Jakub Kicinski and Nicolaas Viljoen. 2016. eBPF Hardware Offload to SmartNICs: cls bpf and XDP. *Proceedings of netdev* 1 (2016).

[56] Abhinav Kommula, Yen-Hung Frank Hu, Mary Ann Hoppa, and Samuel Olatunbosun. 2020. Machine Learning Techniques to Enhance Container Network Security. In *In proceedings of International Conference on Computational Science and Computational Intelligence.* 622–627.

[57] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. 2017. SPEAKER: Split-phase execution of application containers. In *Proceedings of International Conference on Detection of*

*Intrusions and Malware, and Vulnerability Assessment.*

[58] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. 2019. Service mesh: Challenges, state of the art, and future research opportunities. In *Proceedings of IEEE International Conference on Service-Oriented System Engineering*. 122–1225.

[59] Xing Li, Xue Leng, and Yan Chen. 2021. Securing Serverless Computing: Challenges, Solutions, and Opportunities. *arXiv preprint arXiv:2105.12581* (2021).

[60] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. 2018. A measurement study on linux container security: Attacks and countermeasures. In *Proceedings of Annual Computer Security Applications Conference*. 418–429.

[61] Coleman Link, Jesse Sarran, Garegin Grigoryan, Minseok Kwon, M Mustafa Rafique, and Warren R Carithers. 2019. Container Orchestration by Kubernetes for RDMA Networking. In *Proceedings of IEEE International Conference on Network Protocols*. 1–2.

[62] Chang Liu, Longtao He, Gang Xiong, Zigang Cao, and Zhen Li. 2019. Fs-net: A flow sequence network for encrypted traffic classification. In *IEEE INFOCOM 2019-IEEE Conference On Computer Communications*. IEEE, 1171–1179.

[63] Antony Martin, Simone Raponi, Théo Combe, and Roberto Di Pietro. 2018. Docker Ecosystem – Vulnerability Analysis. *Computer Communications* 122 (2018), 30–43.

[64] Jaehyun Nam, Seungsoo Lee, Phillip Porras, Vinod Yegneswaran, and Seungwon Shin. 2022. Secure Inter-Container Communications Using XDP/eBPF. *IEEE/ACM Transactions on Networking* 31, 2 (2022), 934–947.

[65] Jaehyun Nam, Seungsoo Lee, Hyunmin Seo, Phil Porras, Vinod Yegneswaran, and Seungwon Shin. 2020. BASTION: A Security Enforcement Network Stack for Container Networks. In *Proceedings of USENIX Annual Technical Conference*. 81–95.

[66] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. 2019. Flowblaze: Stateful packet processing in hardware. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019*. USENIX ASSOC, 531–547.

[67] Jamal Hadi Salim. 2015. Linux traffic control classifier-action subsystem architecture. *Proceedings of Netdev 0.1* (2015).

[68] Meng Shen, Jinpeng Zhang, Liehuang Zhu, Ke Xu, and Xiaojiang Du. 2021. Accurate decentralized application identification via encrypted traffic analysis using graph neural networks. *IEEE Transactions on Information Forensics and Security* 16 (2021), 2367–2380.

[69] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. 2019. Container Security: Issues, Challenges, and the Road Ahead. *IEEE Access* 7 (2019), 52976–52996.

[70] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. 2018. Security Namespace: Making Linux Security Frameworks Available to Containers. In *Proceedings of USENIX Security Symposium*. 1423–1439.

[71] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. 2018. An analysis and empirical study of container networks. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 189–197.

[72] Linyih Teng, Chi-Hsiang Hung, and Charles H-P Wen. 2022. P4SF: A High-Performance Stateful Firewall on Commodity P4-Programmable Switch. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–5.

[73] Wei Wang, Ming Zhu, Jinlin Wang, Xuewen Zeng, and Zhongzhen Yang. 2017. End-to-end encrypted traffic classification with one-dimensional convolution neural networks. In *2017 IEEE international conference on intelligence and security informatics (ISI)*. IEEE, 43–48.

[74] Jinli Yan, Lu Tang, Junnan Li, Xiangrui Yang, Wei Quan, Hongyi Chen, and Zhigang Sun. 2019. UniSec: a unified security framework with SmartNIC acceleration in public cloud. In *Proceedings of the ACM Turing Celebration Conference-China*. 1–6.

[75] Zirak Zaheer, Hyunseok Chang, Sarit Mukherjee, and Jacobus Van der Merwe. 2019. Eztrust: Network-independent Zero-trust Perimeterization for Microservices. In *Proceedings of the Symposium on SDN Research*. 49–61.