Hyperion: Hardware-Based High-Performance and Secure System for Container Networks

Myoungsung You, Minjae Seo, Jaehan Kim, Seungwon Shin, *Member, IEEE*, and Jaehyun Nam

***Abstract*—Containers have become the predominant virtualization technique for deploying microservices in cloud environments. However, container networking, critical for microservice functionality, often introduces significant overhead and resource consumption, potentially degrading the performance of microservices. This challenge arises from the complexity of the software-based network data plane, responsible for network virtualization and access control within container traffic. To tackle this challenge, we propose `Hyperion`, a novel hardware-based container networking system that prioritizes high performance and security. Leveraging smartNICs, commonly found in cloud environments, `Hyperion` implements a fully-functional container network data plane, encompassing network virtualization and access control. It also has the capability to dynamically optimize its data plane for agile responses to frequent changes in container environments, ensuring up-to-date data plane operation. This hardware-based design empowers `Hyperion` to significantly improve the overall container networking performance without relying on the host system resources. Notably, `Hyperion` seamlessly integrates with existing containerized applications without necessitating modifications. Our evaluation shows that compared to state-of-the-art solutions, `Hyperion` achieves significant improvements in HTTP container communication latency and throughput by up to 2.25x and 4.3x, respectively. Furthermore, it reduces CPU utilization associated with container networking by up to 4x.**

***Index Terms*—Containers, container network optimization, network isolation, network access control, SmartNIC.**

I. Introduction

CONTAINERS [1], have gained significant traction as a viable alternative to virtual machines in cloud environments. This surge in popularity can be attributed to their lightweight virtualization approach, which offers distinct advantages such as improved deployment agility and optimized resource utilization. Containers align well with the microservice architecture [2], where complex applications are decomposed into smaller, more manageable components (i.e., containers). Consequently, the adoption of container technology by a majority of cloud service providers has witnessed rapid growth [1], underscoring its relevance and efficacy in modern cloud computing scenarios.

Real-time communication between containers deployed across multiple hosts is a fundamental requirement in the realm of microservices. As such, the container network plays a crucial role in enabling this communication [3] by virtualizing the underlying physical network infrastructure. This virtualization enables containers to operate as if they were connected to a single shared network, regardless of the complexities of the physical network. However, the current container network architecture introduces notable overhead, which compromises the performance of microservices. Our preliminary evaluation reveals that HTTP communication between containers deployed on different hosts exhibits a substantial increase in latency, up to twofold, compared to direct host communication. Furthermore, this container communication leads to a significant surge in CPU usage of the host, up to 3.5x higher, as elaborated in Section II. Despite the existence of several state-of-the-art container network implementations [4], [5], [6], [7], [8], these problems remain unresolved in both academia and industrial domains.

The root of these problems stems from *two* inherent limitations within the software-based data plane of container networks. First, container traffic traverses *a prolonged forwarding path* involving multiple software components. When a container packet is sent, it first traverses the source container's network stack and then is forwarded to the host network stack. Upon traversal of the host network stack, a series of proxies engage in network virtualization tasks, such as address mapping. Subsequently, the packet is directed to the destination container, undergoing a similar network stack processing sequence. If the destination is situated on a different host, the host network stack encapsulates the packet using tunneling protocols such as VXLAN [9], sending the encapsulated packet to the receiving host. These processes require multiple context-switching and packet copy operations, resulting in increased network latency.

The second limitation pertains to the prevalent use of Layer 7 (L7) protocols (e.g., HTTP) in container communications for microservices [10]. Efficient network virtualization for such containers necessitates L7 traffic processing, including tasks like HTTP path-based routing. Moreover, the connectivity among containers introduces a new attack vector where a malicious container can compromise neighboring containers through various network-based attacks, such as traffic eavesdropping and Man-in-the-Middle (MiTM) attacks [6]. To safeguard container communications, it is imperative for the network data plane

Manuscript received 22 September 2023; revised 1 February 2024; accepted 23 April 2024. Date of publication 20 May 2024; date of current version 6 September 2024. This work was supported in part by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) under Grant IITP-2024-RS-2023-00258649 supervised by the IITP (Institute for Information and Communications Technology Planning and Evaluation). Recommended for acceptance by J. Zhai. *(Corrsponding author: Jaehyun Nam.)*

Myoungsung You, Minjae Seo, Jaehan Kim, and Seungwon Shin are with the School of Electrical Engineering, KAIST, Daejeon 34141, South Korea (e-mail: famous77@kaist.ac.kr; ms4060@kaist.ac.kr; jaehan@kaist.ac.kr; claude@kaist.ac.kr).

Jaehyun Nam is with the Department of Computer Engineering, Dankook University, Yongin 16890, South Korea (e-mail: jaehyun.nam@dankook.ac.kr).

Digital Object Identifier 10.1109/TCC.2024.3403175

2168-7161 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on October 04,2024 at 05:47:44 UTC from IEEE Xplore. Restrictions apply.

to scrutinize L7 content within container traffic, like HTTP headers, and block unauthorized L7 accesses among containers. However, the current data plane relies on software proxies to deliver L7 packet processing capabilities, including L7 routing and L7 inspections. This *software proxy-based design* requires redundant network stack traversals and packet copy operations. For example, container packets should be routed to user-space proxies from the kernel after traversing the host network stack. Additionally, these proxies consume many system resources when performing payload analysis and matching regular expressions against the analyzed payload. These overheads degrade networking performance and place a substantial burden on the host's CPU (Section VI).

Here, the key question is how to build an optimized container network data plane that achieves high-performance and secure communication, minimizes CPU utilization, and maintains compatibility with containerized applications.

Our approach. In this work, we thus propose `Hyperion`, a novel hardware-based container networking system designed to tackle the aforementioned limitation by leveraging a programmable hardware Network Interface Card (e.g., a smartNIC). To the best of our knowledge, this is the first endeavor toward achieving a complete hardware implementation of container networking. `Hyperion` implements a high-performance and secure network data plane tailored for containers on top of a smartNIC. It facilitates direct packet exchange among containers via the smartNIC, bypassing intermediate software stacks, which are the main performance bottlenecks, and establishing a streamlined traffic forwarding path. Upon packet arrival, hardware engines within the smartNIC deliver all of the network functions required for container networking, including virtualization and load balancing. Simultaneously, these engines inspect packet headers and payloads based on security policies, isolating traffic flows between irrelevant containers to ensure secure communication. Moreover, `Hyperion` incorporates a feedback-based optimization mechanism that dynamically re-configures its data plane in response to the related container events. This ensures up-to-date data plane operation and maintains line-rate performance even in the face of frequent changes within container environments. By offloading container networking to the smartNIC, `Hyperion` achieves remarkable performance improvements without imposing a burden on host system resources. Furthermore, this hardware-based design ensures full compatibility with existing containerized applications.

Contribution. We make the following contributions:

- We comprehensively analyze today's container network implementations, focusing on profiling their performance overhead and identifying the main bottleneck.
- We design a novel hardware-based container networking system, `Hyperion`, which facilitates high-performance and secure networking without relying on host resources.
- We provide a full implementation of `Hyperion` using commodity smartNIC [11] and extensively evaluate `Hyperion` using realistic workloads. The results demonstrate that `Hyperion` achieves up to a 2x improvement in TCP latency and up to a 2.25x reduction in HTTP latency compared to state-of-the-art solutions.

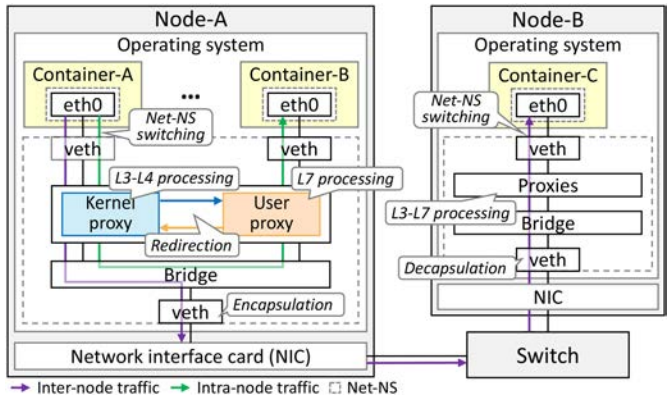


Fig. 1. A common container network architecture across multiple nodes.

II. Background and Challenges

We begin by explaining the container network architecture, which enables communication between containers across multiple hosts. Through real-world examples, we then demonstrate a significant adverse performance impact on microservices in the current container network architecture.

A. Container Network Architecture

Fig. 1 depicts the container network architecture adopted by modern container platforms, such as Kubernetes [12] and Openshift [13]. Container platforms create a container cluster, which consists of a set of hosts (referred to as nodes), and then assign containers to individual nodes. Containers execute small modular units for microservices and communicate with each other via the underlying container network to process user requests. Each container operates within its own isolated network environment, called a network namespace (net-NS) [14]. This net-NS has a dedicated network interface (`eth0` in a container) and a network stack, ensuring that a container cannot directly access network interfaces or traffic of other containers. Similarly, a node also has its net-NS, known as the host net-NS, with its network interface (`eth0` in the host) connected to the physical network interface card. To facilitate communication between containers residing in different net-NSs, container platforms utilize a bridge network. As shown in Fig. 1, a container is connected to a bridge interface located on the host net-NS through a virtual interface (`veth`) [15], which is paired with the container's interface. Consequently, packets from the container are forwarded to the host net-NS via the `veth` pair and sent to their destination using the bridge interface. This architecture enables communication between containers across multiple nodes, promoting isolation and network connectivity for containerized microservices.

Service Communication. Containers have a short lifetime, meaning that they can be terminated or updated at any time, causing their IP addresses to fluctuate. This creates a challenge for other containers needing to communicate directly with them. To address this issue, container platforms employ a service abstraction [16], which offers a consistent virtual IP address (i.e., a service IP address) linked to a group of containers. Unlike

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on October 04,2024 at 05:47:44 UTC from IEEE Xplore. Restrictions apply.

container IP addresses, a service IP address remains stable even when the associated containers (e.g., endpoints) are replaced, ensuring a reliable endpoint for accessing the application hosted on those containers. Moreover, the service abstraction facilitates traffic distribution across multiple containers serving the same application. This load-balancing mechanism optimizes resource utilization and enhances the application's availability by efficiently distributing incoming traffic loads.

Packet Journey through the Container Network. When a container sends packets, they first traverse from the container's net-NS to the host net-NS through the `veth` pair, known as net-NS switching. In the host net-NS, the packets are handled by both kernel- and user-level proxies, as shown in Fig. 1. These proxies are responsible for delivering the necessary network functions for container communication. The kernel-level proxy is implemented using eBPF [17] and iptables [18], enabling efficient Layer 3 to Layer 4 processing. Its primary task is to map the service IP addresses within the packets to the corresponding endpoint. By doing so, this proxy selects an appropriate endpoint for the given packets. Then, it rewrites their destination IP addresses and ports to match the selected endpoint. Container platforms implement network access control to enhance network security and thwart potential network attacks between containers. After mapping service IP addresses, the kernel-level proxy inspects the packet headers and filters out any packets that violate predefined policies (e.g., packets using disallowed TCP ports). Subsequently, the inspected packets are forwarded to the user-level proxy, which operates in the user space and focuses on providing application layer services for container packets. This user-level proxy [19] begins by scrutinizing the contents of the packet payloads and discards those that violate L7 policies, such as attempting to access unauthorized HTTP URLs. After that, the packets are returned to the kernel-level proxy for final handling.

Upon inspection, the kernel-level proxy forwards the packets to the bridge interface, allowing them to be received by the destination container's interface, as shown by green arrows in Fig. 1. If the destination container is located on a different node, additional processing is required since network switches cannot route packets with virtual and private IP addresses of containers. In such a scenario, the kernel-level proxy employs packet encapsulation [9], wherein it adds an extra header containing the source and destination node's IP addresses to the original packet, making the packet routable by network switches. Subsequently, the packet is transmitted to the underlying network and received by the target node, where it is processed in reverse order, as depicted by purple arrows in Fig. 1.

B. Challenges in Current Container Networks

C1. Time-consuming Packet Forwarding Path: The current data plane for container networking comprises various and loosely coupled software components, including `veth` pairs, a set of proxies, and the bridge interface. This design creates a time-consuming packet forwarding path between these components and incurs a number of packet copies, net-NS switching, and redundant network stack processing during container packet processing. As a result, there is a significant increase in context switching and software interrupts, degrading the performance of microservices.

TABLE I
HTTP LATENCY AND THROUGHPUT UNDER A LOAD OF 30 K RPS (VALUES IN PARENTHESES ARE STANDARD DEVIATIONS)

Scenario	Latency (μs)	Throughput (RPS)
Host-network (baseline)	1.22 (0.63)	171K
Container	1.61 (0.73), +31%	74K, -56%
Container (L7 inspection)	2.68 (2.02), +119%	33K, -80%

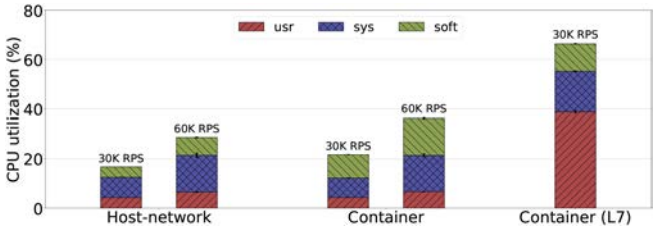


Fig. 2. CPU utilization of a server-side node under a load of 30 K and 60 K RPS. The CPU utilization is divided into three categories: user-space (`usr`), inside the kernel-space (`sys`), and software interrupt handling (`soft`).

To assess this performance issue, we established a container cluster by connecting two physical nodes equipped with Intel Xeon 40-cores and a 40GbE hardware NIC. We employed Cilium [5], a cutting-edge container network implementation. Within this setup, we deployed a web server (nginx) and a client container on each node and proceeded to measure the HTTP latency and throughput between them utilizing `wrk2`. To measure the CPU utilization on the server-side node during the experiment, we utilized `mpstat`. As a baseline, we also conducted identical tests in a host-network mode, where containers communicate directly through the host IP addresses with the container network virtualization. Table I shows the adverse impact of the prolonged packet forwarding path on the performance of the containerized web service. Compared to the host-network mode, container communication experiences a substantial decline in HTTP throughput by 56% (down to 74 K) and a 31% increase in latency (to 1.61 μs). Moreover, the inefficient data plane design significantly raises CPU utilization. Under the same HTTP load of 60 K requests per second (RPS), container communication demands about 38% more CPU resources than the host-network mode, as illustrated in Fig. 2. These performance degradation and increases in CPU resources primarily result from the creation of extra software interrupts (`soft`) stemming from the time-consuming packet forwarding path and numerous packet copies between various data plane components. Specifically, the container network setting generates twice as many software interrupts as the host-network mode.

C2. Software Proxy-based L7 Processing: The current data plane for container networks faces another challenge concerning the enforcement of L7 policies. Given that containers for microservices typically communicate over L7 protocols (e.g., HTTP REST APIs), most container platforms support L7 routing (or load balancing) and L7 security policies by default to ensure

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on October 04,2024 at 05:47:44 UTC from IEEE Xplore. Restrictions apply.

efficient and secure container communication. However, even state-of-the-art solutions encounter considerable overhead when enforcing L7 policies. The impact of this overhead is demonstrated in Table I, where a single L7 security policy, which limits the client container's access to a specific URL (*test-*.html*), leads to a significant drop in throughput from 74 K to 33 K. Also, latency increases by 119% to 2.68 ms, and CPU utilization rises to 65% (see Fig. 2), more than double the previous measurement.

The primary reason behind this performance issue lies in the software proxy-based approach adopted by current container network data planes. This approach requires packet redirection to the user-level proxy for payload analysis with regular expressing matching and subsequently returning the packets back to the previous kernel-level proxy. While this design allows L7 processing tasks (e.g., L7 inspection), unsuitable for kernel-level processing, to be offloaded to a user-level proxy, it introduces additional overhead due to the extra packet redirection between the two proxies. Also, payload inspection itself is a resource-intensive and time-consuming task for software implementations. Hence, the CPU cycles consumed by user space processes significantly escalate, reaching up to 39% of the total CPU utilization as shown in Fig. 2. This substantial increase in CPU utilization leads to degraded overall network performance, impacting the efficiency of containerized services.

III. Hyperion Overview

A. Assumptions and Threat Model

Assumptions. In this work, we make the explicit assumption that the underlying infrastructure, encompassing the node (host OS) and its hardware (including smartNICs), is trustworthy and enjoys secure protection of privileges. However, we acknowledge that containers running microservices may pose potential security risks. As demonstrated in previous studies [20], [21], [22], [23], adversaries can exploit vulnerabilities in container images and containerized applications, compromising these containers. In the same context, we consider the possibility of adversaries leveraging these compromised containers as a stepping stone for launching further attacks, such as lateral movement within the container network.

Threat Model. Similar to prior studies [6], [20], our primary security objective revolves around safeguarding containers from network attacks perpetrated by adversaries (e.g., TCP session hijacking). We aim to establish a least-privileged inter-container communication environment where containers are restricted from transmitting malicious packets that violate pre-defined security policies. Our focus is also preventing unauthorized access to cluster resources, such as other containers and services, through the container network. Our main security goal is to ensure the security of container communication by inspecting container traffic. Therefore, we exclude system-based attacks (e.g., container escape [24]), which exploit vulnerabilities in the kernel and applications. Such attacks are typically mitigated in prior studies [25], [26] by limiting system access (e.g., system calls) from containers. Additionally, we do not address resource exhaustion attacks (e.g., DoS) perpetrated by malicious containers. Container platforms are already equipped with efficient control mechanisms [27] (e.g., cgroups) to regulate network resources allocated to containers.

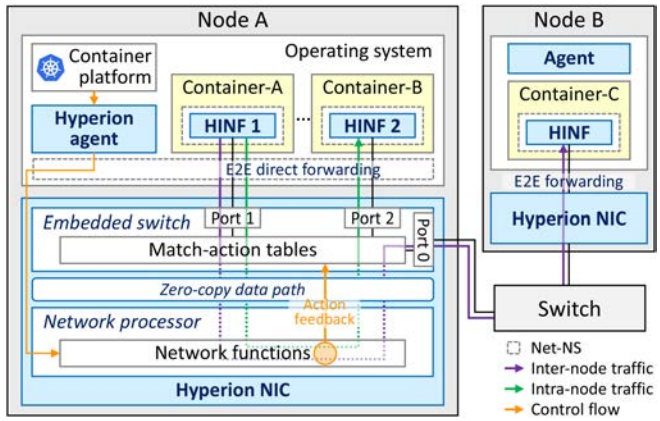


Fig. 3. The overall architecture of Hyperion system.

B. Design Overview

To address the limitations of current software-based container networks, we propose Hyperion, a novel hardware-based system that leverages a smartNIC [28], [29], implementing the entire data plane for containers at the hardware layer. In particular, commodity smartNICs are equipped with hardware engines capable of parallel packet processing and a network processor for supporting various network-related tasks. However, implementing the container network data plane on a smartNIC while maintaining line-rate performance poses significant challenges due to the limited programmability of hardware engines and the restricted performance capabilities of the network processor. To address these limitations, we re-design the container network data plane, considering the unique attributes of the hardware components within smartNICs. In addition, we introduce two optimization techniques (i.e., E2E forwarding and action feedback) for our new hardware data plane. These techniques enable the data plane to not only facilitate efficient packet exchange with containers (Section IV-B1) but also to maintain line-rate performance, even under frequent changes in container environments (Section IV-C3).

Fig. 3 illustrates the overall architecture of Hyperion. It comprises two key components: Hyperion agent and Hyperion NIC (HNIC), both deployed on each node within the container cluster. The agent, integrated within the container platform, monitors the network states of active containers through a standard interface [30] and dynamically updates the HNIC configuration in response to container changes. In Hyperion, container packets are managed via a specialized virtual interface, the Hyperion interface (HINF). This interface allows containers to exchange packets with the HNIC in the same way as using traditional veth interfaces. This HINF also implements a direct and efficient packet forwarding path between containers and the HNIC, improving the overall network performance.

When receiving a packet, the embedded switch within the HNIC handles L3/L4 network functionalities, including network virtualization and L3/L4 security control. Packets requiring L7

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on October 04,2024 at 05:47:44 UTC from IEEE Xplore. Restrictions apply.

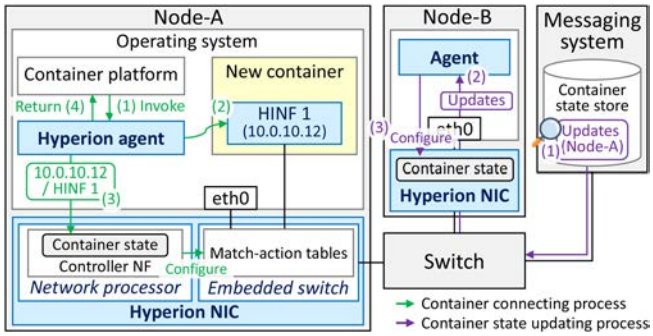


Fig. 4. The agent is responsible for connecting containers to the HNIC (green) and updating its policies (purple).

processing are designated as *exception packets* and are directed to the network processor through a zero-copy data path. The network processor is responsible for executing advanced network functions (e.g., L7 routing, load balancing) and inspecting packet payloads for malicious content. After that, these packets are returned to the embedded switch, with the processing results instantaneously reflected back to the embedded switch (*Action Feedback* in Fig. 3). This feedback mechanism effectively reduces the number of exception packets, thereby optimizing the packet processing performance of the HNIC. Finally, the embedded switch routes the packets to their designated destinations, either to another container's HINF (intra-node communication) or externally via the network wire (inter-node communication). All these processes are accelerated by dedicated hardware units within the HNIC, delivering high-performance container networking while conserving host system resources for microservice operation.

IV. Hyperion Internals

This section is dedicated to a detailed discussion of each component of `Hyperion`.

A. Hyperion Agent

A `Hyperion` agent serves as a crucial component within the `Hyperion` system, operating as a daemon process on each node. It is responsible for managing the hardware-based data plane (the HNIC) by performing two vital functions. 1) It creates a direct channel between containers and the HNIC. 2) In addition, it dynamically updates the HINC's policies in response to relevant container events.

1) Seamless Communication Channel: The primary role of a `Hyperion` agent is to establish connections between containers on a node and the HNIC. For this, the agent is designed to operate on top of the standard interface [30] provided by container platforms; thus, it seamlessly integrates with existing container environments. The connection process involves four distinct steps, as depicted by the green arrows in Fig. 4. Upon the creation of a new container, 1) the `Hyperion` agent is invoked by container platforms to configure the network for the new container. 2) The agent deploys the `Hyperion` interface (HINF), which directly links to the HNIC, in place of the veth and assigns a network address to the HINF from the container address range assigned to the node. Subsequently, 3) the agent updates the new container's context, including its IP address and HINF ID, to the HNIC, enabling the HNIC to recognize and process packets from the container. With the successful configuration, the container can now efficiently communicate with others through the HNIC. Finally, 4) the agent notifies the container platform that the newly created container is connected and ready for use, providing seamless integration and continuous management. Upon termination of a container, the agent withdraws the assigned HINF and removes its context from the HNIC, ensuring proper cleanup and resource management.

It is important to note that the `Hyperion` agent operates seamlessly on top of the standard interface offered by container platforms. The agent only makes changes to the network interface within a container when connecting it to our system. This design approach avoids any modifications to container platforms and containerized applications.

2) HNIC Configuration Aligned With Container Changes: Another critical responsibility of the `Hyperion` agent is to dynamically update policies within the HNIC in response to container events. As mentioned earlier, the agent can promptly detect container events that occur within its node from container platforms. However, to ensure seamless and consistent operation of essential data plane services for container communication, such as routing and service load balancing, it becomes imperative for each agent's state to be continuously updated in response to container events occurring across different nodes. To achieve this, the `Hyperion` agent employs an event-driven state update mechanism, leveraging existing messaging systems in the cluster. Container platforms typically employ a messaging system, such as etcd, as a robust and persistent storage solution for all API data and resource states. This storage includes vital information about the cluster's configuration and state, such as active containers, nodes, and service status. Hence, by leveraging this messaging system, the state update process of the `Hyperion` agent is methodically organized into a three-step procedure to maintain an up-to-date and accurate view of the cluster's resources for optimal data plane service provision, as depicted by the green arrows in Fig. 4.

1) Each agent monitors the messaging system, utilizing APIs provided by the container platforms, to detect relevant state updates. Upon identifying updates concerning containers operating on its node, 2) the agent retrieves the updated data from the messaging system and transmits it to its controller. Subsequently, 3) network functions and match-action tables in the HNIC are reconfigured in alignment with the newly updated container state. This state-updating mechanism synchronizes container states across each HNIC in the cluster, ensuring that each HNIC maintains a consistent perspective of cluster resources.

B. Hardware-Based Container Networking

The HNIC operates as a hardware-based data plane responsible for processing all container traffic originating from or entering a node. As depicted in Fig. 5, the embedded switch carries out the core packet processing, ensuring line-rate performance.

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on October 04,2024 at 05:47:44 UTC from IEEE Xplore. Restrictions apply.

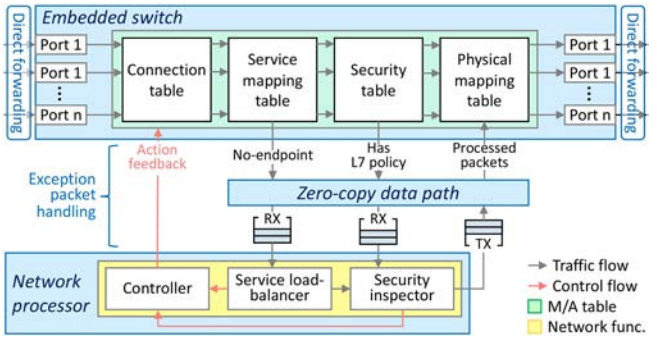


Fig. 5. The hardware-based container packet processing.

When packets require more detailed control than what the switch provides, they are marked as exception packets and passed to the network processor for specialized handling. The processing results of these exception packets are fed back to the embedded switch on the fly, enabling efficient handling of subsequent packets associated with the same flows. In this section, we delve into the details of the embedded switch, which is meticulously designed to facilitate line-rate container networking, utilizing five fundamental techniques to optimize its performance.

1) Packet Forwarding Path Optimization: In the current container networking setup, packets are forwarded to their destination containers through veths and the bridge interface, which resides in different net-NSs. However, this complex forwarding path introduces unnecessary overheads, such as net-NS switching and host network stack processing, potentially affecting overall performance. To address this concern and optimize the forwarding path, we introduce `Hyperion` interface (HINF), which are virtual functions (VFs) of Single Root I/O Virtualization (SR-IOV) [31], replacing the veths. Packets originating from a HINF follow a streamlined path, bypassing the host net-NS and all intermediate stacks, and are directly forwarded to the smartNIC's RX buffer through the SR-IOV data path. Fig. 5 illustrates how packets enter the dedicated port assigned to the source HINF and then undergo processing by the embedded switch. After completing the switch processing, packets are either routed to the destination container's HINF via the SR-IOV data path or transmitted to the underlying network. This end-to-end direct forwarding significantly simplifies the packet forwarding path between containers and the HNIC, improving the overall performance of inter-container networking.

2) Inter-Container Connection Tracking: Upon end-to-end direct forwarding, packets undergo processing by the embedded switch, offering line-rate network virtualization for containers. The embedded switch is equipped with a specialized set of match-action tables, specifically designed for container communication, as illustrated in Fig. 5. To ensure efficient container communication, the connection context between the source and destination containers is taken into account. Thus, the first table, the connection table, is designed to monitor network connections (sessions) between containers. Each entry in the connection table comprises a 5-tuple representing the connection's characteristics and relevant contexts, such as states and SYN/ACKs, for an active network connection. Upon the arrival of a new packet, the HNIC updates the corresponding table entry for the incoming packet based on its header information and stores the connection context in the packet's metadata for subsequent pipeline stages. In the case of a packet associated with a new connection, a fresh table entry is added. The HNIC removes connection table entries when closing packets are identified or if no subsequent packets are received within a specified timeout, preventing table entry explosion.

3) Service Address Mapping: As described in Section II, containers typically communicate using a service address (e.g., Cluster IP [16] in Kubernetes) rather than the IP address associated with each container. Thus, the data plane for container networks must be capable of mapping the service IP address within a packet to the appropriate endpoint container, effectively directing the packet to its destination. To achieve this functionality, the embedded switch incorporates a second table specifically designed to handle the service address mapping process.

The service mapping table maintained by the embedded switch is responsible for storing information about the endpoints (container IP addresses and ports) of each service. When a packet arrives, the HNIC extracts the endpoint container information from this table based on the connection context stored in the packet metadata. Then, it performs Destination Network Address Translation (DNAT) by replacing the packet's destination IP address and port number with those of the endpoint container. This mapping is performed oppositely on the receiving end, known as Source Network Address Translation (SNAT). This address mapping distributes traffic to endpoints for the same service, effectively achieving load-balancing and improving resource utilization. It is important to note that the network processor updates the service mapping table in response to relevant events, such as the creation of a new endpoint container. If incoming packets are not associated with a pre-defined endpoint yet, they are passed to the network processor via the zero-copy datapath (labeled as *no-endpoint* in Fig. 5), ensuring accurate address mapping. Eventually, the selected endpoints for exception packets are dynamically stored in this table on the fly, enabling the switch to perform address mapping for subsequent packets at the line rate.

4) Network Isolation: In multi-tenant cloud environments, where different tenant containers operate on shared nodes, it is crucial to ensure network security by restricting network visibility and communication scope for containers. Our security model assumes that containers within the same microservice trust each other, whereas containers in different microservices may not. To enforce this security model, the switch's third table blocks unauthorized access between containers belonging to different microservices while enabling fine-grained control of traffic flow between containers within the same microservice. By carefully managing the rules in this table, `Hyperion` effectively isolates containers from different microservices, enhancing network security and preventing potential security breaches.

The security table in the embedded switch stores information about active microservices in the cluster and the security policies applicable to containers within those microservices. Using this table, the switch first verifies if a packet's source

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on October 04,2024 at 05:47:44 UTC from IEEE Xplore. Restrictions apply.

and destination belong to the same microservice. In the case of inter-microservice packets, they are immediately dropped, except when specific policies defined by an operator permit such communication. This filtering effectively isolates each microservice into a dedicated logical network segment, preventing the lateral movement of adversaries and reducing the potential blast radius of security breaches. Subsequently, the embedded switch examines if the packet complies with the security policies within the microservice. Out-of-policy packets, which attempt to use prohibited ports or protocols, are discarded to prevent network attacks between containers. During policy checking, if Layer 7 policies exist, packets are passed to the network processor for payload inspection (labeled as *L7 policy* in Fig. 5). Our system leverages the connection context to avoid redundant inspection. Hence, packets from an already-established connection bypass this security table, streamlining the inspection process and optimizing performance.

5) Physical Address Mapping: The final table of the embedded switch in our `Hyperion` system is designed for physical address mapping, which is crucial in container networks where containers utilize virtual and private IP addresses. The physical mapping table stores the network address (physical address) of each node and the container address range (e.g., 10.244.0.0/24) assigned to that node. Using this table, the HNIC first identifies the node associated with the destination container of a packet. Subsequently, it creates a virtual header based on the identified node's physical address and inserts this virtual header before the existing packet headers. This encapsulation process allows the packet to be forwarded to the destination node by network switches. Upon reaching the receiving node, these encapsulated packets are decapsulated and relayed to their respective destination containers.

It is essential to note that the HNIC can further enhance the security of inter-node communication by encrypting the encapsulated packets (e.g., using IPSec with pre-shared keys). This encryption ensures secure communication between nodes and mitigates the risk of potential traffic eavesdropping by adversaries. These packet processing tasks are efficiently accelerated by dedicated hardware logic within the HNIC, guaranteeing line-rate inter-node communication. For intra-node packets, on the other hand, direct routing to the destination container's HINF is employed, eliminating the need for address mapping or encapsulation and further optimizing communication within the node.

C. Supporting Fine-Grained Data Plane Services

The HNIC is designed to provide fine-grained data plane services through three advanced network functions running on its network processor.

During the embedded switch's processing, exception packets are passed to the relevant network functions running on the network processor for further handling. This exception packet handling ensures that complex tasks that cannot be executed within the embedded switch are offloaded to the network processor. In this work, a zero-copy data path is utilized to achieve fast communication between the embedded switch and network

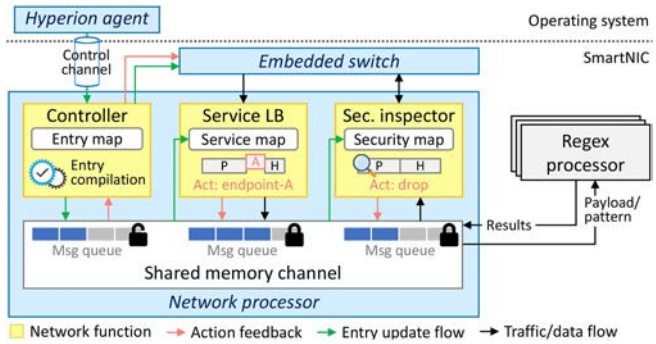


Fig. 6. Network functions running on the HNIC.

functions, as shown in the center of Fig. 5. This enables the direct transfer of exception packets from the switch's buffer to the dedicated RX queue of a network function, and vice versa. By optimizing data exchanges between the switch and network functions, `Hyperion` ensures that the HNIC maintains high performance during frequent exception packets.

Packets from the embedded switch undergo processing by a set of network functions, including a service load-balancer and a security inspector. Each network function offers necessary data plane services for exception packets by analyzing headers and payloads, and forwards the processed packets along with metadata to the next function. As shown in Fig. 6, packets and data exchanges between functions are conducted through shared memory channels, operating on a wait-notify basis. When a sender function has new data to transmit, it first attempts to acquire a lock on the destination function's message queue. Upon acquisition, the sender inserts a data pointer into the message queue and signals the destination that the data is ready. Subsequently, the destination retrieves the data from its queue for processing. This shared memory channel eliminates data shipping costs between functions, such as copying and (de)serialization, thereby enhancing processing latency.

Finally, the processed packets are directed to the last table in the switch for the final routing decision and physical address mapping via the zero-copy data path (*processed packets* in Fig. 5). Also, the processing results of exception packets are stored as appropriate table rules by the controller function (*action feedback* in Fig. 5), enabling the swtich to handle subsequent packets at line-rate without involving exception handling. We now detail the network functions provided by our system.

1) Service Load-Balancing: The service load-balancer function integrated into the network processor is designed to seamlessly align with the service abstraction offered by container platforms (e.g., Kubernetes). It efficiently distributes service traffic across multiple endpoint containers, maximizing resource utilization and mitigating the risk of overwhelming any endpoint. Leveraging the capabilities of the smartNIC, our load-balancer function implements an optimized load-balancing scheme without requiring a separate software proxy to run on the host OS. This hardware-based approach enhances performance and reduces resource overhead, resulting in efficient and scalable load balancing for containerized applications.

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on October 04,2024 at 05:47:44 UTC from IEEE Xplore. Restrictions apply.

As shown in Fig. 6, the service load-balancer function in Hyperion utilizes a service map containing information about the endpoints (container IP addresses and ports) and their predefined weights for each service. This service map is managed by both the controller and the agent, and it is dynamically updated on the fly whenever there are changes to a service or its endpoints (see Section IV-A2). When a packet arrives, the load-balancer function first consults the service map. It selects a suitable endpoint using a load-balancing algorithm, which can be round-robin, weighted round-robin, hashing, or other algorithms based on the characteristics of service workloads. After selecting an endpoint, the function rewrites the packet's destination IP address and port to match the chosen endpoint. The selected endpoint and the packet's connection state are then transmitted to the controller via the shared memory queue (the red arrow in Fig. 6) and inserted into the service mapping table. This dynamic feedback mechanism enhances load-balancing performance by reducing the number of exception packets, especially in microservice scenarios where containers frequently reuse long-lasting connections. Also, it ensures that packets belonging to the same connection are consistently directed to the same endpoint, establishing connection affinity. Finally, the processed packets are forwarded to the following network function through the shared memory.

2) Application-Aware Network Access Control: The security inspector function in Hyperion addresses the challenge of enforcing restrictions on application layer (L7) communication between containers in microservice environments without incurring significant CPU consumption or degrading network performance. Instead of relying on user-level proxies, which can be resource-intensive, the security inspector offloads L7 inspection tasks to the smartNIC. When an exception packet arrives, the security inspector retrieves the relevant L7 security policies and detection patterns for the packet from the security policy map. It then performs an in-depth inspection of the packet's content, including headers, URL paths, and methods, and matches it against the retrieved L7 policies.

To achieve high-performance L7 inspection, the security inspector leverages a regular expression (regex) processor, a common hardware accelerator found in commodity smartNICs. This regex processor conducts hardware-based parallel pattern matching on behalf of the network processor, which may have limited performance capabilities. The security inspector is designed to interact with the regex processor through dedicated queues, allowing efficient data transfer and pattern matching. The regex processor dequeues packet payloads and patterns from these queues and inserts matching results. Based on these results, the security inspector determines if the packet complies with the L7 policies. The security inspector then sends benign packets back to the embedded switch via the zero-copy datapath, while it immediately discards out-of-policy packets.

If configured, the security inspector can also send a response packet containing an appropriate status code (e.g., 403 error) to the sender container after discarding the original packet. Furthermore, the processing results, such as dropping a specific flow, can be stored in the security table, enabling early identification and dropping of subsequent packets from the same malicious flow. It is important to note that the security map includes the contents of the security table, allowing for the enforcement of network isolation against packets received from the service load-balancer on behalf of the embedded switch's security table. This comprehensive approach to L7 inspection within the smartNIC ensures efficient and secure container communication without compromising performance or resource utilization.

3) Event-Driven Data Plane Policy Management: The controller function is responsible for managing data plane policies stored in the match-action tables and network function maps in response to container events, ensuring that the data plane operates with the most up-to-date information. There are two scenarios in which table and map updates are triggered. i) Agent-initiated updates: Triggered when the controller receives updates from the agent about container changes on the node, as shown by green arrows in Fig. 6. In this scenario, both match-action tables and maps are updated based on the information from the agent. ii) Function-initiated updates: Triggered when the controller receives new actions for exception packets from other functions, shown by red arrows. Here, only the relevant entries in the embedded switch's table require updates.

The controller function is designed to support efficient policy updates. A Direct Memory Access (DMA)-based control channel is utilized for streamlined interaction with the agent. Upon initialization, the agent and controller negotiate a DMA-compatible memory area to establish an event-driven control channel over this area. This channel, eschewing traditional packet-based channels, avoids CPU-intensive tasks like network stack traversals, facilitating low-latency communication between the agent and the controller function. Furthermore, the controller function supports concurrent updates for multiple table entries, enabling rapid deployment of data plane policies. Each controller instance runs on a separate core of the network processor, along with its own update queue. Updated entries are dispatched to these instances based on their content, and the dispatch results are sorted into the entry map. This ensures consistent assignment of specific entries to the same instance, with each instance submitting updates for its entries to the hardware via its unique queue. This design eliminates the need for locking the data structures for entries and allows the use of multiple queues when interacting with the hardware, optimizing the efficiency of policy updates.

V. Implementation

We have implemented a full prototype of Hyperion utilizing the Nvidia Bluefield 2 [32], a smartNIC widely found in cloud environments. In this section, we will delineate the details of each component of the prototype.

Hardware Component Bluefield 2 DPU is equipped with a hardware packet processing ASIC, multiple programmable ARM cores, and a regex accelerator. The hardware packet processing ASIC provides line-rate packet matching and modification primitives. We implemented the match-action pipeline of the HNIC by utilizing these primitives. Additionally, we configured the containers' SR-IOV Virtual Functions (VF) to directly interact with the packet processing ASIC for efficient

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on October 04,2024 at 05:47:44 UTC from IEEE Xplore. Restrictions apply.

packet exchange. Our network functions are developed with a total of approximately 4,000 lines of C code, leveraging DPDK [33] and DOCA APIs [34]. These functions are instantiated in multiple threads, each anchored to individual ARM cores. Data exchanges among these functions are efficiently conducted through shared memory queues, developed using the Boost Interprocess Library [35]. For L7 processing, these functions offload payload-matching tasks to the regex accelerator, employing DOCA DPI APIs. We have also addressed and rectified certain bugs in the current DPI APIs related to hardware-based flow controls, thereby further enhancing overall performance. The controller function, crucial in runtime management, dynamically updates the offloaded table rules (such as match and action) within the ASIC, based on the analysis of exception packets. It accomplishes this through the use of multiple update queues, enabling concurrent interactions with the ASIC. The controller function also maintains communication with the Hyperion agent on the host through DOCA DMA channels for seamless policy updates.

Software Component The agent, developed as a Container Network Interface (CNI) [30] plugin, operates on the host OS. It integrates with container platforms, facilitating the connection of newly instantiated containers to the HNIC, as well as the disconnection of terminated containers from the HNIC. Each agent persistently monitors the state of container platforms utilizing Kubernetes APIs [36]. Any modifications to cluster resources (e.g., containers) are tracked by the agent, which is able to access this pertinent information from the cluster storage (e.g., etcd). Upon receiving updated information, the agent transmits this data to the controller via the DOCA comm channel APIs.

VI. Evaluation

Here, we conduct a comprehensive evaluation of Hyperion focusing on three key dimensions. 1) We first compare the end-to-end performance of Hyperion with two state-of-the-art solutions and analyze its performance characteristics through a microbenchmark. 2) We then delve into the network security model of Hyperion, employing real-world attack scenarios. 3) Lastly, we analyze the compatibility of Hyperion with existing container platforms.

A. Experimental Environment

To evaluate the effectiveness of Hyperion, we establish a real-world testbed by connecting two physical nodes. Each node is equipped with an Intel Xeon Silver 4114 CPU (40 cores), 64 GB of memory, and runs the Ubuntu 22.04 OS. These nodes have Bluefield2 DPUs, including the Hyperion prototype, and are directly connected via a 40GbE cable to remove the interference from intermediate devices (e.g., switches). We deploy Kubernetes on these nodes, with Docker serving as the container runtime.

B. Performance Evaluation

First, we evaluate Hyperion with two popular containerized applications: Nginx (webserver) and Memcached (key-value store). To evaluate end-to-end performance, we measure the performance between the server and client containers under the enforcement of security policies that restrict communication between them based on predefined conditions, such as HTTP URL and key prefix. Across performance experiments, we conduct comparative evaluations of Hyperion against two popular solutions: Cilium [5] and Istio [37]. Cilium, one of the fastest CNIs, uses an eBPF-based L3/L4 data path and a node-level proxy [19] for L7 processing. Istio, the leading solution in microservice networking, processes container traffic via per-container proxies. Also, we employ a vanilla Kubernetes using Flannel CNI [38] with no security inspection as a baseline.

TABLE II
HTTP Throughput Measurements

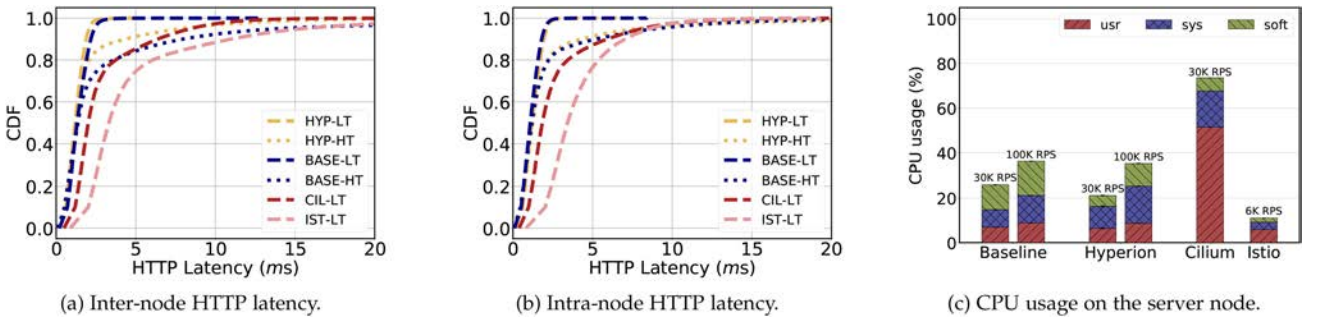
Throughput (RPS)	*Baseline*	**Hyperion**	*Cilium*	*Istio*
Intra-node	151.2K	140.7K (-7%)	32.5K (-79%)	8.1K (-94%)
Inter-node	117.5K	114.3K (-3%)	31.4K (-74%)	6.2K (-94%)

Hyperion surpasses cilium and istio by 4x and 17x times, respectively, and shows performance comparable to the baseline.

1) Nginx performance: To measure the performance of Nginx, we deploy two containers into our testbed. The server container runs Nginx v1.25, while the client container runs the widely-used HTTP benchmark tool, wrk2. This evaluation focuses on evaluating Hyperion under various HTTP traffic load scenarios for a comprehensive performance analysis. Accordingly, we configure the benchmark tool to generate 30 K and 100 K HTTP requests per second (RPS) to simulate low (LT)- and high-traffic (HT) load scenarios. In both scenarios, we set the server's HTTP response size to 1 K bytes for uniformity. We deploy server and client containers across two different nodes in the inter-node setup. Conversely, in the intra-node setup, these containers are co-located within the same node. To ensure secure communication between the server and client, we enforce security policies that confine their HTTP communication to a specific TCP port (80), an HTTP method (GET), and a URL prefix (nginxperf-*.html).

As shown in Table II, Hyperion outperforms other solutions in HTTP throughput and achieves performance on par with the baseline. Specifically, Hyperion processes HTTP requests at rates 4.3x and 3.6x higher than Cilium in inter-node and intra-node setups, respectively, and exhibits a 17x throughput increase over Istio. When compared to the baseline, Cilium and Istio show significant throughput degradations of 79% and 94%, respectively. In contrast, Hyperion maintains a mere 7% reduction in throughput, while conducting comprehensive L3 to L7 security inspections. Further, Hyperion demonstrates notable enhancements in HTTP latency, as shown in Fig. 7(a) and (b). Under low traffic conditions, Hyperion maintains an average latency 2.25x faster than Cilium. In tail latency, Hyperion achieves a P99 latency of approximately 2.5 ms, significantly lower than Cilium's 12.4 ms. Istio, unable to handle 30 K RPS traffic (see Table II), is evaluated under a 6 K RPS load (its maximum load), where Hyperion still outperforms it in both average and tail latency. The superior performance of Hyperion becomes even more pronounced

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on October 04,2024 at 05:47:44 UTC from IEEE Xplore. Restrictions apply.



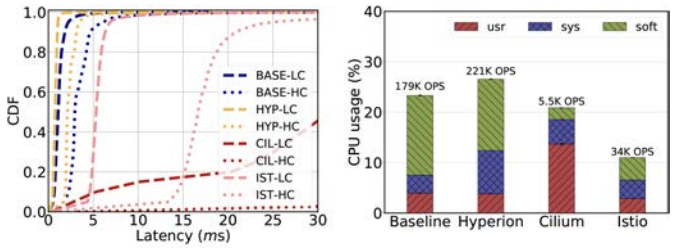
(a) Inter-node HTTP latency. (b) Intra-node HTTP latency. (c) CPU usage on the server node.

Fig. 7. Performance measurement of Nginx. Due to their limited processing capability, Cilium and Istio cannot be evaluated in the high-traffic load scenario (HT). Hyperion outperforms Cilium by an average of 2x while using 4x less CPU.

under HT scenarios. At 100 K RPS, Hyperion matches the baseline performance and achieves up to 1.8 times faster average latency in inter-node setups. The evaluation of Cilium and Istio under this HT condition is omitted due to their limited throughput capacities. This performance enhancement is attributed to our hardware-based design, which contrasts with the software-based approaches of Cilium and Istio. Cilium's performance is hindered by its node-level proxy, and Istio's per-container proxy approach exacerbates this degradation. By offloading resource-intensive tasks to the smartNIC, Hyperion achieves superior performance unattainable by software-based solutions.

CPU Usage. In our analysis of CPU resource consumption, we employed the mpstat tool to monitor CPU usage on the server-side node. Under a 30 K RPS HTTP traffic load, Cilium exhibits a CPU usage rate of 73%, with its user-level proxy consuming about 50% of CPU resources (usr) for payload inspection. Despite its limited processing capacity of 6 K RPS, Istio still consumes considerable CPU usage due to its per-container proxy approach. In contrast, Hyperion demonstrates a significantly lower CPU usage compared to Cilium, with a fourfold reduction, and maintains similar CPU usage to the baseline in both low- and high-traffic load scenarios. This reduced CPU consumption further underscores Hyperion's efficiency in processing container traffic.

2) Memcached performance: Next, we measure the performance of Memcached, a widely employed key-value store for microservices. For this evaluation, we distribute a Memcached server container and a Memtier_benchmark client container across two different nodes. This evaluation also centers on evaluating the connection-level scalability of Hyperion by varying the number of concurrent connections. Specifically, we configure the benchmark tool to generate 200 concurrent connections in a low-connection load scenario (plots with *-LC) and 600 concurrent connections in a high-connection load scenario (plots with *-HC). We configure a GET and SET operation ratio of 5:5, and the data size is configured to 1 k bytes. Furthermore, we enforce security policies that restrict Memcached communication with a specific TCP port number (11211) and a key prefix (memtierbench-*). Note that Istio is evaluated without the key prefix matching due to its limited support for Memcached protocols. For brevity, we omit the intra-node scenario for this evaluation, as its results align closely with those of the inter-node scenario.



(a) Inter-node latency. (b) CPU usage on the server.

Fig. 8. Comparison of Memcached performance on the inter-node setting. Istio is evaluated without the key prefix matching due to its limited functionality.

Hyperion exhibits significant improvements in Memcached latency between the server and client containers. Fig. 8(a) shows the CDF of combined latency for both GET and SET operations in the two connection load scenarios. Under the low-connection load, Hyperion shows an average latency of 0.75 ms, a substantial 48-fold enhancement over Cilium's 36.2 ms, and remains comparable to the baseline. Even against Istio, which skips the expensive key prefix matching, Hyperion exhibits a 7.3x faster average latency. This performance disparity is attributed to Istio's design, where its per-container proxy processes all traffic at the user space, leading to redundant network stack processing and packet copies regardless of the key prefix matching. In high-connection load scenarios, Hyperion's average latency stands at 2.39 ms, outperforming both Cilium and Istio by substantial margins. Notably, Hyperion excels in tail latency (P99) within high-connection loads, showing a twofold improvement over the baseline. These results affirm Hyperion's ability to sustain high performance in securing container communication, even under high connection load.

CPU Usage. Fig. 8(b) presents the CPU usage on the server-side node during this evaluation. Unlike the previous experiment, the benchmark tool did not support testing under a fixed Memcached traffic rate. Therefore, our focus is on measuring the server's CPU usage under low-connection load scenarios, where Cilium and Istio reached their maximum throughput. While Cilium and Istio exhibit lower CPU usage, Hyperion processes 40x and 6.5x more Memcached traffic than Cilium

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on October 04,2024 at 05:47:44 UTC from IEEE Xplore. Restrictions apply.

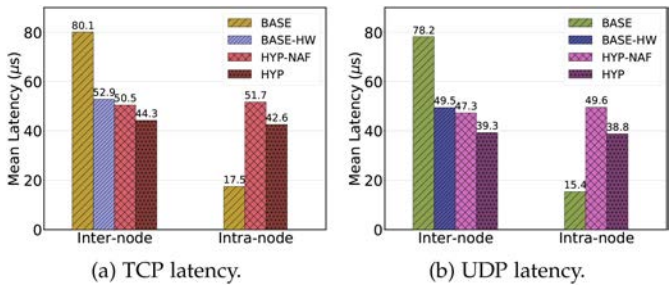


Fig. 9. TCP/UDP latency comparison of Hyperion and the baseline setting. The bar plots with BASE-HW and HYP-NAF mean that the baseline with VXLAN offloading and Hyperion without the auto-feedback mechanism.

TABLE III
TCP THROUGHPUT COMPARISON OF Hyperion AND THE BASELINE WITH VARYING CONCURRENT TCP CONNECTIONS

TCP throughput	*Inter-node*		*Intra-node*	
	1-conn	*2-conn*	*1-conn*	*2-conn*
Baseline (VXLAN offload)	13.5 Gb/s 20.4 Gb/s	22.7 Gb/s 36.5 Gb/s	20.9 Gb/s	41.3 Gb/s
Hyperion	20.1 Gb/s	36.5 Gb/s	18.7 Gb/s	39.6 Gb/s

and Istio, respectively. This marked improvement is attributed to Hyperion's hardware-based design, as discussed in the previous evaluation.

3) Microbenchmark: In the context of Hyperion, wherein a container's interface is replaced with a HINF and all container traffic is processed through the smartNIC, potential overhead may emerge during intra-node communication. Specifically, this design requires copying intra-node container traffic to the smartNIC's memory from the kernel space, leading to potential network latency increments. To evaluate the extent of this overhead, we conduct a comparative analysis between Hyperion and a plain Kubernetes with the Flannel CNI [38]. A hardware baseline is also utilized. This baseline configuration involves offloading the packet (de)encapsulation tasks (VXLAN) to the smartNIC, while other networking tasks are managed using Flannel CNI. This evaluation focuses on measuring the round-trip latency and throughput of TCP communication. During our evaluation, all security inspections of Hyperion are disabled, allowing us to concentrate specifically on assessing the impact of traffic forwarding overhead.

Fig. 9 provides a comparison of the average latency between Hyperion and the baseline. Under the inter-node scenario, Hyperion leads to notable improvements in average latency up to 44%. As shown in Table III, Hyperion also achieves near maximum TCP throughput with only two concurrent connections. In both latency and throughput metrics, Hyperion exhibits performance on par with the hardware baseline. This enhancement primarily originates from the offloading of container networking tasks, such as service IP management and packet encapsulation, to the smartNIC. Additionally, Hyperion is uniquely capable of directly routing inter-node traffic from the smartNIC to the destination container's network namespace (net-NS), effectively bypassing the host's net-NS. This approach

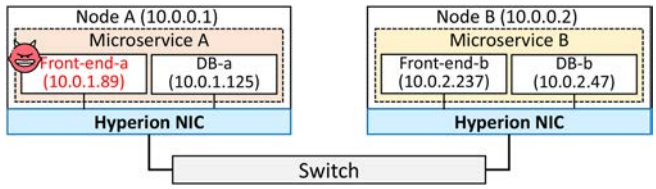


Fig. 10. Security evaluation environments.

eliminates unnecessary traversals of the network stack, thereby reducing latency. Moreover, the auto-feedback mechanism further enhances Hyperion's latency by up to 17% by reducing the number of exception packets (see HYP-NAF).

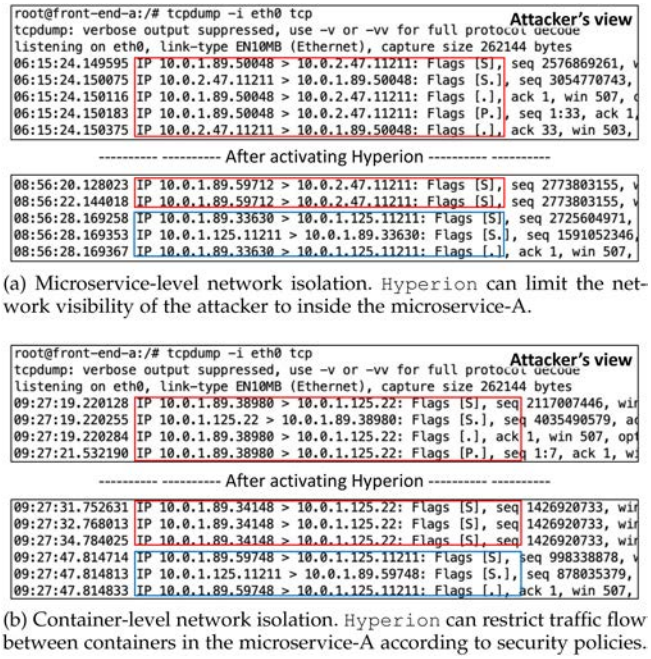
However, a contrasting result emerges in the intra-node scenario, where Hyperion encounters up to 23.4us latency overhead. Additionally, it exhibits a slightly lower TCP throughput compared to the baseline. This overhead's origin lies in Hyperion's hardware-based design. In conventional container networks, intra-node container traffic does not involve smartNIC processing and is directly forwarded to the destination container via virtual interfaces. Contrarily, Hyperion requires the forwarding of such traffic to the smartNIC through the PCIe interface. This process creates extra packet copying operations between host memory and the smartNIC, traversing the host's PCIe stack. Consequently, Hyperion's hardware-based design can incur some overhead during intra-node container communication. Nonetheless, this design choice concurrently provides substantial performance enhancements and efficient resource utilization for securing container communication. Considering the benefits of Hyperion underscored by prior evaluations, this micro-second scale overhead in inter-node communication remains negligible.

C. Security Evaluation

In this evaluation, we demonstrate the effectiveness of Hyperion in securing container networking. As shown in Fig. 10, we deploy two microservices, which consist of a front-end server (Nginx) and a DB container (Memcached), on our testbed. We assign the label -a to containers of microservice-A and the label -b to those of microservice-B. We assume that the front-end-a container is compromised by an attacker and aims to move to other containers.

1) Network-Level Access Control: Here, we focus on validating the effectiveness of Hyperion against unauthorized network access between containers. Ideally, network visibility and traffic flow between containers should be confined within the boundary of each respective microservice. Without Hyperion, the malicious container (front-end-a) can initiate a network connection to the DB-b container, which is associated with another microservice, as shown in the upper part of Fig. 11(a). This absence of network isolation between microservices allows the attacker to identify all of the containers within the cluster and execute various network attacks to compromise them further. However, upon activation of Hyperion, the network visibility and traffic flow are isolated within the boundary of each microservice. As shown in the bottom of Fig. 11(a), all

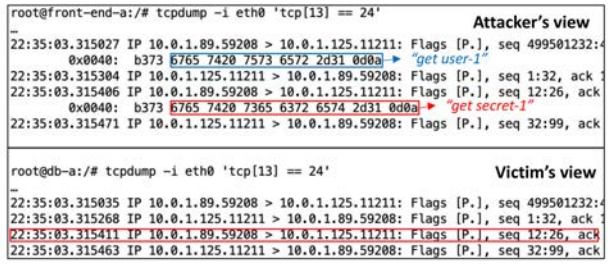
Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on October 04,2024 at 05:47:44 UTC from IEEE Xplore. Restrictions apply.



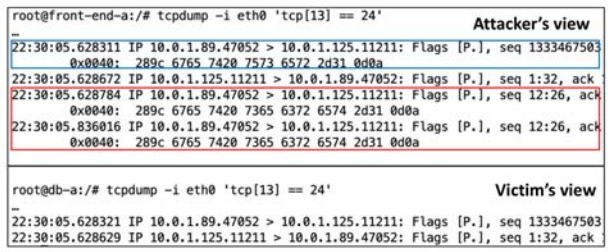
(a) Microservice-level network isolation. Hyperion can limit the network visibility of the attacker to inside the microservice-A.

(b) Container-level network isolation. Hyperion can restrict traffic flow between containers in the microservice-A according to security policies.

Fig. 11. The effectiveness of Hyperion's network isolation.



(a) Without Hyperion, the attacker (IP: 10.0.1.89) can access all contents, including those labeled with the *secret-** key prefix, stored in the DB-a container (IP: 10.0.1.125).



(b) With Hyperion, packets containing the *secret-** key prefix, sent by the attacker, are discarded and not forwarded to the DB-a container.

Fig. 12. Hyperion protects Memcached communication of the DB-a container by inspecting packet payloads.

of the packets sent to the DB-b container from the attacker are discarded by Hyperion. Consequently, the attacker's capacity to identify and access containers within different microservices is nullified.

Beyond the isolation of microservices, it is equally crucial to enforce fine-grained network access control within a microservice to enhance container network security. Unrestricted communication within a microservice creates opportunities for attackers to compromise other containers within the same microservice. Consider the upper section of Fig. 11(b), where the front-end-a container (the attacker) connects to the DB-a container via the SSH port, typically reserved for administrators. Without Hyperion, this scenario exposes sensitive database content to potential risks. However, with Hyperion active, fine-grained access control is enforced at the level of individual containers, as depicted in the lower part of Fig. 11(b). According to the policy, Hyperion blocks the attacker's SSH access while permitting the use of TCP port 11211, following the Memcached protocol. These evaluations show Hyperion's capabilities in isolating traffic across microservices and controlling container communication within a microservice.

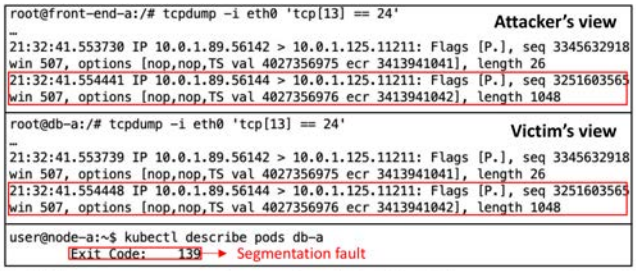
2) Application-Level Access Control: Next, we move to show the effectiveness of Hyperion in restricting L7 communication between containers. In the absence of Hyperion, the front-end-a container enjoys unrestricted access to all the contents within the DB-a container, including those marked with the *secret-** key prefix, as illustrated in Fig. 12(a). This unrestricted L7 access creates a potential risk of data leakage. Upon activation, Hyperion enforces access control on Memcached (L7) communication between these two containers by scrutinizing the packet payload. As depicted in Fig. 12(b), Hyperion performs an in-depth analysis of Memcached protocol fields. It selectively discards packets containing disallowed key prefixes (*secret-*), while permitting those with benign ones (*user-*). Consequently, Hyperion effectively confines the attacker's L7 access to the DB-a container, thereby introducing a more granular security layer atop its network-level access control.

In addition to the L7 access control, Hyperion possesses the capability to identify and eliminate malicious content within the general packet payload, thus safeguarding containerized applications against known vulnerability exploitation. Without Hyperion, the attacker can exploit a known buffer overflow vulnerability (CVE-2016-8704) [39] within the DB-a container. As illustrated in Fig. 13(a), this attack is achieved by transmitting malicious Memcached packets containing an exploit code composed of an excessive amount of "A"s. As shown in the bottom part of Fig. 13(a), this attack results in the termination of the DB-a container, signaled by exit code 138, indicative of an unexpected termination due to a segmentation fault. It's worth noting that while we have omitted further details of potential exploits stemming from this vulnerability for brevity, it can lead to remote code execution, thereby granting the attacker full control over the victim container. Hyperion effectively mitigates such attacks through a comprehensive examination of packet payloads. As depicted in Fig. 13(b), Hyperion discards malicious Memcached packets that contain excessively long value fields, while selectively permitting benign packets to be forwarded to the DB-a container. This capability empowers Hyperion to function as a comprehensive network security solution that effectively neutralizes potential threats within containerized applications.

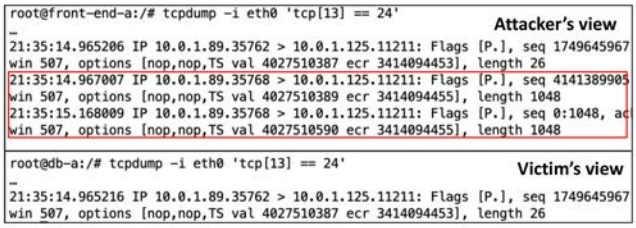
D. Compatibility Analysis

Lastly, we analyze the compatibility of Hyperion with container platforms to demonstrate its ability to integrate

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on October 04,2024 at 05:47:44 UTC from IEEE Xplore. Restrictions apply.

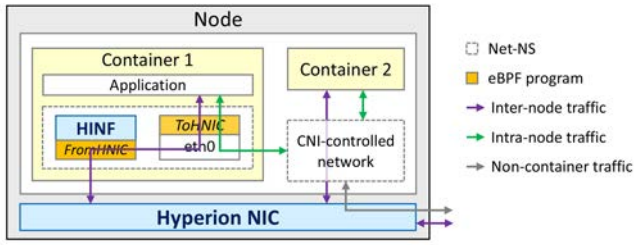


(a) Without Hyperion, the attacker (IP: 10.0.1.89) can trigger the vulnerability of the DB-a container (IP: 10.0.1.125) by sending malicious Memcached packets containing an exploit payload.

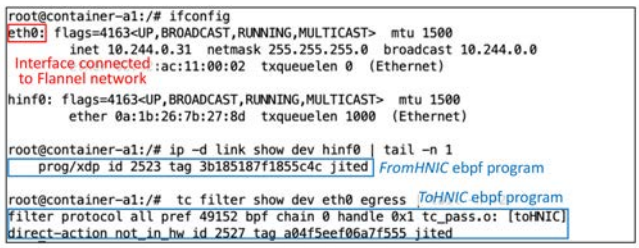


(b) Hyperion discards Memcached packets containing exploit payloads and prevents them from being forwarded to the DB-a container.

Fig. 13. Hyperion blocks malicious packets designed to exploit the victim's vulnerability [39].



(a) The overview of the plugin mode. Container traffic is forwarded to the HNIC or the CNI-controlled network based on pre-defined policies.



(b) A real-world example of the plugin mode with Flannel CNI.

Fig. 14. Hyperion supports the plugin mode that operates as an extension of a main CNI.

transparently. Container platforms rely on CNIs like Flannel [38] and Calico [4] for network configuration. Hyperion is designed with seamless integration with these CNIs in mind, supporting two operational modes: i) a standalone mode for optimized performance and ii) a plugin mode for enhanced traffic management flexibility.

1) Standalone Mode: In standalone (default) mode, Hyperion functions as a standard CNI [40], accelerating TCP/UDP-based container communication through its hardware data plane, the HNIC. When a container is instantiated, the Hyperion agent configures the container's network environment and connects it to other containers. This involves installing a HINF in the container's net-NS and assigning a network address based on an IPAM plugin [41] or predefined configurations. Consequently, all container communications are efficiently handled by the HNIC at the hardware level. As discussed in Section IV-A1, Hyperion merely replaces network interfaces of containers to facilitate connection to the HNIC, ensuring transparent compatibility with container platforms.

2) Plugin Mode: In plugin mode, Hyperion operates as an extension for a main CNI (e.g., Flannel). Fig. 14(a) shows this mode's architecture, where the main CNI manages container network configurations, and the Hyperion agent connects new containers to the HNIC post main CNI operation. Notably, the original network interface (eth0) remains active alongside the deployed HINF. Two eBPF programs (fromHNIC and toHNIC) are attached to these interfaces, redirecting traffic between these interfaces based on pre-defined policies. This setup allows selective traffic processing by the HNIC while maintaining connectivity via the original interface (eth0), offering more flexibility than the standalone mode.

Fig. 14(b) shows a real-world example with Flannel as the main CNI. In this setup, an application uses eth0 and remains unaware of hinf0. However, the attached eBPF programs enable inter-node traffic to be processed by the HNIC, while intra-node communication utilizes the CNI-controlled network via eth0. The toHNIC program redirects inter-node traffic sent by the application from eth0 to hinf0, which is then forwarded to the HNIC. Conversely, inter-node traffic from the HNIC is received at hinf0 and redirected to eth0, allowing the application to receive it normally. This plugin mode does not necessitate modifications to applications or the main CNI, as it transparently redirects traffic at the interface level. Despite potential redirection overhead, our preliminary evaluation indicates it is negligible (less than 10 μs), primarily due to the eBPF layer's efficient traffic handling before network stack processing.

VII. Related Work

Container Networking. As containers have become widely adopted for cloud environments, several studies [42], [43] have analyzed the performance characteristics of container networks and identified bottleneck points.

To address this problem, various studies have emerged in both industry [4], [5] and academia [6], [8]. Cilium [5] and Bastion [6], for example, employ an eBPF-based fast data plane for container networks, replacing the slow centralized bridge interface with fast per-container eBPF-based network stacks. Despite the efforts made in previous software-based studies, they continue to face challenges in terms of performance and resource utilization, especially when dealing with L7 container packets, as detailed in Section II. In contrast, our system takes a hardware-based approach, effectively relieving the host CPU from the burden of container packet processing and achieving high-performance container networking.

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on October 04,2024 at 05:47:44 UTC from IEEE Xplore. Restrictions apply.

Hardware Offloading. Recently, researchers have proposed various systems [44], [45], [46] capable of offloading CPU-intensive packet processing tasks (e.g., TCP connection processing) onto dedicated hardware to enhance network performance in cloud environments. For instance, systems such as NICA [44] and MECaNIC [45] facilitate tenant applications (e.g., VMs) to offload such tasks onto smartNICs while ensuring performance and resource isolation among them. However, most of these studies are designed for VM-based cloud environments. They cannot be deployed directly in container environments, where multiple containers sharing the same kernel communicate through a virtualized network that connects their isolated namespace boundary. For instance, MECaNIC [45] is incapable of direct packet exchange with containers. In addition, these studies focus on L3/L4 processing and lack L7 processing capabilities, which can be a significant drawback in container environments. In contrast, `Hyperion` is designed for container-based cloud environments. It implements the high-performance and full-fledged data plane equipped with L3 to L7 container packet processing capabilities, utilizing a SoC-based smartNIC.

Container Communication Acceleration. Some recent literature has explored the adoption of Remote Direct Memory Access (RDMA) in cloud environments to accelerate container communication [47], [48], [49]. For instance, PipeDevice [49] employs RDMA for intra-node container communication, improving bulky data transfer between containers on the same host. While these studies contribute significantly to the field, they focus on RDMA communication and do not address the broader, more prevalent scope of general TCP/IP communication within container networks. In contrast, `Hyperion` is designed to accelerate general TCP/IP container communication across both inter- and intra-host scenarios, making it more versatile and applicable to a majority of container networks. Furthermore, enabling containerized applications to use RDMA necessitates code-level modifications to both container platforms (e.g., Kubernetes) and applications. This is due to the fact that containerized applications, by nature of their virtualized stack (e.g., namespaces), cannot directly interact with RNICs. In contrast, `Hyperion` eliminates the need for such modifications and maintains compatibility with upper-layer software, offering more general deployment scenarios.

VIII. Discussion and Limitations

While `Hyperion` addresses the limitations of existing solutions and improves container networking performance, there remains much room for further enhancements.

Scalability. The scalability of `Hyperion` is limited by the hardware resources available within the underlying smartNIC. In our current implementation, the maximum number of table entries supported by one HNIC in line-rate is set at 8,000 entries. This limit can be extended through the utilization of the smartNIC's external memory (up to 16 GB). The number of containers that can be linked to one HNIC is limited by the number of VFs the smartNIC supports. The current implementation can support up to 254 containers on a single node. Considering that container platforms can support up to 110 containers (or container groups sharing a single namespace) on a single node [50], our current system implementation can be effectively deployed in typical cloud environments without encountering scalability problems.

PCI bus overhead. `Hyperion` handles container traffic through the use of a smartNIC. A drawback of this design is that even intra-node traffic should be passed to the smartNIC via the PCI bus, potentially incurring extra overhead in intra-node communication scenarios. As demonstrated in Section VI, such micro second-level overheads are negligible in comparison to the benefits derived from offloading CPU-intensive container packet processing tasks (e.g., L7 inspection) to the smartNIC. Moreover, this limitation can be mitigated by optimizing the datapath between the containers and the smartNIC using zero-copy and kernel bypass techniques [33]. Incorporating these techniques into `Hyperion` is an avenue we intend to explore in future work.

IX. Conclusion

In this work, we have introduced `Hyperion`, a novel smartNIC-based high-performance data plane tailored for container environments. `Hyperion` has effectively addressed the limitations of software-based container networking by offloading CPU-intensive container packet processing tasks to a smartNIC. `Hyperion` has seamlessly integrated with existing containerized applications and underlying container platforms without requiring modifications. Comprehensive evaluations conducted under realistic workloads have also demonstrated that `Hyperion` outperforms state-of-the-art solutions up to 4x in container communication. We believe that `Hyperion` is poised to be a practical and pioneering approach in the growing prevalence of hardware-assisted cloud networking.

References

[1] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE Trans. Cloud Comput.*, vol. 7, no. 3, pp. 677–692, Third Quarter, 2019.
[2] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure devops," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2016, pp. 202–211.
[3] Y. Park, H. Yang, and Y. Kim, "Performance analysis of CNI (container networking interface) based container network," in *Proc. Int. Conf. Inf. Commun. Technol. Convergence*, 2018, pp. 248–250.
[4] "Project calico," 2013. [Online]. Available: https://www.projectcalico.org/
[5] "Cilium," 2017. [Online]. Available: https://www.cilium.io/
[6] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin, "BASTION: A security enforcement network stack for container networks," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 81–95.
[7] D. Zhuo et al., "Slim: OS kernel support for a low-overhead container overlay network," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation*, 2019, pp. 331–344.
[8] J. Lei, M. Munikar, K. Suo, H. Lu, and J. Rao, "Parallelizing packet processing in container overlay networks," in *Proc. 16th Eur. Conf. Comput. Syst.*, 2021, pp. 261–276.
[9] "Cilium overlay network," 2017. [Online]. Available: https://docs.cilium.io/en/v1.9/concepts/networking/routing/
[10] M. Grambow, L. Meusel, E. Wittern, and D. Bermbach, "Benchmarking microservice performance: A pattern-based approach," in *Proc. Annu. ACM Symp. Appl. Comput.*, 2020, pp. 232–241.
[11] "Mellanox bluefiled2 DP," 2021. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf
[12] "Kubernetes," 2024. [Online]. Available: https://kubernetes.io
[13] "RedHat OpenShift," [Online]. Available: https://www.openshift.com
[14] "Namespaces in operation, part 1: Namespaces overview," 2013. [Online]. Available: https://LWN.net/

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on October 04,2024 at 05:47:44 UTC from IEEE Xplore. Restrictions apply.

[15] "Veth–virtual ethernet device," 2023. [Online]. Available: https://man7.org/linux/man-pages/man4/veth.4.html
[16] "Service – kubernetes," 2024. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/
[17] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 1–36, 2020.
[18] "Netfilter and IPtables," 2024. [Online]. Available: https://www.netfilter.org
[19] "Cilium envoy extension," 2020. [Online]. Available: https://docs.cilium.io/en/v1.9/concepts/security/proxy/envoy/
[20] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE Access*, vol. 7, pp. 52976–52996, 2019.
[21] K. Brady, S. Moon, T. Nguyen, and J. Coffman, "Docker container security in cloud computing," in In *Proc. Annu. Comput. Commun. Workshop Conf.*, 2020, pp. 975–980.
[22] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A measurement study on linux container security: Attacks and countermeasures," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 418–429.
[23] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, "Docker ecosystem–vulnerability analysis," *Comput. Commun.*, vol. 122, pp. 30–43, 2018.
[24] Z. Jian and L. Chen, "A defense method against docker escape attack," in *Proc. Int. Conf. Cryptogr., Secur. Privacy*, 2017, pp. 142–146.
[25] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *Proc. 23rd Int. Symp. Res. Attacks, Intrusions Defenses*, 2020, pp. 443–458.
[26] L. Lei et al., "Speaker: Split-phase execution of application containers," in *Proc. Detection Intrusions Malware, Vulnerability Assessment: 14th Int. Conf.*, Bonn, Germany, Springer, 2017, pp. 230–251.
[27] R. Rosen, "Resource management: Linux kernel namespaces and cgroups," *Haifux*, vol. 186, 2013, Art. no. 70.
[28] "NVIDIA blue field 2 DPU," 2021. [Online]. Available: https://resources.nvidia.com/en-us-accelerated-networking-resource-library/bluefield-2-dpu-datasheet?lx=LbHvpR&topic=networking-cloud
[29] "Marvell OCTEON 10 DPU," 2023. [Online]. Available: https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-octeon-10-dpu-platform-product-brief.pdf
[30] "CNI: The container network interface," 2024. [Online]. Available: https://www.cni.dev/
[31] "PCI-SIG single root I/O virtualization (SR-IOV) support in intel virtualization technology for connectivity," 2008. [Online]. Available: https://www.intel.com/content/www/us/en/pci-express/pci-sig-single-root-io-virtualization-support-in-virtualization-technology-for-connectivity-paper.html
[32] "NVIDIA bluefield-2 DPU," 2021. [Online]. Available: https://www.nvidia.com/en-us/networking/products/data-processing-unit/
[33] "Data plane development kit," 2012. [Online]. Available: https://www.intel.com/content/www/us/en/communications/data-plane-development-kit.html
[34] "NVIDIA DOCA library APIs," 2024. [Online]. Available: https://docs.nvidia.com/doca/sdk/doca-libraries-api/index.html
[35] "Boost interprocess," 2016. [Online]. Available: https://www.boost.org/doc/libs/1_63_0/doc/html/interprocess.htm
[36] "Kubernetes API Reference," 2024. [Online]. Available: https://kubernetes.io/docs/reference/
[37] ""Istio service mesh," 2024. [Online]. Available: https://istio.io/latest/
[38] "Flannel CNI," [Online]. Available: https://github.com/flannel-io/flannel
[39] "CVE-2016–8704 detail," 2016. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2016--8704/
[40] "Kubernetes SR-IOV plugin," 2024. [Online]. Available: https://github.com/k8snetworkplumbingwg/sriov-network-device-plugin/
[41] "Kubernetes CNI plugins," 2024. [Online]. Available: https://github.com/containernetworking/plugins/
[42] K. Suo, Y. Shi, A. Lee, and S. Baidya, "Characterizing networking performance and interrupt overhead of container overlay networks," in *Proc. ACM Southeast Conf.*, 2021, pp. 93–99.
[43] S. Qi, S. G. Kulkarni, and K. Ramakrishnan, "Assessing container network interface plugins: Functionality, performance, and scalability," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 1, pp. 656–671, Mar. 2021.
[44] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein, "NICA: An infrastructure for inline acceleration of network applications," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 345–362.
[45] T. Park, M. You, J. Cui, Y. Jin, K. Lee, and S. Shin, "MecaNIC: Smartnic to assist URLLC processing in multi-access edge computing platforms," in *Proc. IEEE 30th Int. Conf. Netw. Protoc.*, 2022, pp. 1–12.
[46] S. Grant, A. Yelam, M. Bland, and A. C. Snoeren, "Smartnic performance isolation with fairnic: Programmable networking for the cloud," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl., technol., Architectures, Protoc. Comput. Commun.*, 2020, pp. 681–693.
[47] T. Yu, S. A. Noghabi, S. Raindel, H. Liu, J. Padhye, and V. Sekar, "FreeFlow: High performance container networking," in *Proc. 15th ACM Workshop Hot Topics Netw.*, 2016, pp. 43–49.
[48] Y. Zhang, Y. Tan, B. Stephens, and M. Chowdhury, "Justitia: Software multi-tenancy in hardware kernel-bypass networks," in *Proc. 19th USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 1307–1326.
[49] Q. Su et al., "PipeDevice: A hardware-software co-design approach to intra-host container communication," in *Proc. 18th Int. Conf. Emerg. Netw. EXperiments Technol.*, 2022, pp. 126–139.
[50] "Kubernetes: Considerations for large clusters," 2024. [Online]. Available: https://kubernetes.io/docs/setup/best-practices/cluster-large/

Myoungsung You received the BS degree in computer science from Chungbuk National University, and the MS degree from the Graduate School of Information Security, KAIST. He is working toward the PhD degree with the School of Electrical Engineering, KAIST. His research interests include programmable network data planes, cloud security, and distributed systems.

Minjae Seo received the BS degree in computer engineering from Mississippi State University, and the MS degree from the Graduate School of Information Security, KAIST. He is a researcher with KAIST, Daejeon, South Korea. His current research interests include Software-defined networking security, network fingerprinting, and deep learning-based network system.

Jaehan Kim received the BS and MS degrees from the School of Electrical Engineering, KAIST. He is working toward the PhD degree with the School of Electrical Engineering, KAIST. His current research interests mainly focus on cyber security, machine learning Security, large language model and data mining.

Seungwon Shin (Member, IEEE) received the BS and MS degrees from KAIST, both in electrical and computer engineering, and the PhD degree in computer engineering from the Electrical and Computer Engineering Department, Texas A&M University. He is an associate professor with the School of Electrical Engineering, KAIST. He is currently a vice president with Samsung Electronics, leading the security team in the IT and Mobile Communications Division. His research interests span the areas of Software-defined networking security, DarkWeb analysis, and Cyber threat intelligence.

Jaehyun Nam received the BS degree in computer science and engineering from Sogang University, Korea, and the MS and PhD degrees from the School of Computing (Information Security), KAIST. He is an assistant professor with the Department of Computer Engineering, Dankook University, Korea. His research interests focus on networked systems and security. He is especially interested in security issues in cloud and edge computing systems (including SDN/NFV, IoT, and container security).

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on October 04,2024 at 05:47:44 UTC from IEEE Xplore. Restrictions apply.