HardWhale: A Hardware-isolated Network Security Enforcement System for Cloud Environments

Myoungsung You*, Jaehyun Nam†, Hyunmin Seo*, Minjae Seo*, Jaehan Kim*, Dongmin Choi*, Seungwon Shin*
*KAIST, †Dankook University

2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS) | 979-8-3503-8605-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDCS60910.2024.00053

***Abstract*—With the increasing popularity of containers for deploying microservices, ensuring the security of container networks has become a vital concern. However, current security solutions rely on a host's operating system (OS) to enforce network policies for container traffic. This design incurs severe overhead and cannot guarantee container network security when attackers gain access to the host's OS. Therefore, we propose *HardWhale*, a hardware-isolated network security enforcement system for containers that delivers high-performance and robust network security without depending on the host's OS. *HardWhale* leverages a smartNIC, physically isolating the entire container traffic inspection stack from the host and accelerating inspection tasks. Inspection policies securely reside within the smartNIC and are updated in runtime without involving the host, due to our isolated policy management mechanism. This design ensures robust network security for containers, even if the host is exposed to attackers. Evaluations show that *HardWhale* protects containers against various network attacks in compromised environments and improves HTTP throughput threefold and HTTP latency 2.3-fold compared to state-of-the-art solutions.**

I. Introduction

In recent years, containers [1] have gained widespread adoption in cloud environments as an alternative to virtual machines. This is due to their ability to support the agile deployment of microservices, a new architecture in which a complex application is built as a set of small and independent modules (containers). However, the increased use of containers has also led to increased container security incidents. A recent survey [2] found that 94% of respondents reported experiencing at least one container security incident in the past year. These incidents are often attributed to the shared kernel model of containers, which allows a malicious container to access the kernel and compromise co-located containers. Researchers have developed various solutions [3], [4] to prevent malicious system access initiated by containers, such as Confine [3] that generates policies for restricting a container's system call invocations.

Despite this focus on hardening individual containers, there has been limited research on securing container communications. In particular, the use of containers as microservices, in which containers are used to create complex cloud services, requires real-time *network* communication between containers distributed across multiple hosts. This network connectivity between containers has become a new attack vector to gradually compromise the entire cloud system. One such attack is lateral movement, in which attackers who have compromised one container can move within the cloud system by compromising other containers through several network attacks [5], [6]. Lateral movement is a key tactic in advanced persistent threats, allowing attackers to infiltrate deep inside the cloud system and exfiltrate sensitive data [7].

Unfortunately, current security solutions for container network [5], [8]–[11] are unable to prevent lateral movement as they still rely on a host's operating system (OS) to inspect container traffic and enforce network policies. They suffer from two fundamental limitations. First, they cannot prevent lateral movement when attackers gain access to the host's OS, which commonly happens in the real world (e.g., container escape [7], [12]). Second, these solutions consume significant system resources (e.g., CPU) to inspect traffic sent by hundreds of containers running on the host, noticeably degrading the performance of microservices.

Our approach. This paper introduces *HardWhale*, a novel hardware-isolated container network security system aimed to address the limitations of existing solutions. *HardWhale* deploys robust and high-performance container network security stacks on each host in the container system by leveraging a smartNIC. Situated on a smartNIC, the network security stack operates independently from the host, providing two essential functions for container network security: *security policy enforcement* and *policy management*. By offloading these functions to the trusted smartNIC, *HardWhale* establishes a physical isolation layer that prevents potential attackers within the host from compromising the effectiveness of these two functions.

All container packets, whether originating from the host or arriving from the network wire, are controlled by *HardWhale*. It enforces comprehensive policies against these packets, analyzing both their headers and payloads, and blocks unauthorized traffic flow between containers at all network layers. Hardware-based security engines are utilized for this inspection, offering line-rate performance. Additionally, *HardWhale* ensures identity verification for every container packet transmitted from each host, preventing unauthorized traffic inflow. Policies for inspection are securely stored within the smartNIC and dynamically managed in response to changes in container environments. *HardWhale* updates policies through a secure update channel without involving the host's OS, ensuring policy enforce-

2575-8411/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDCS60910.2024.00053

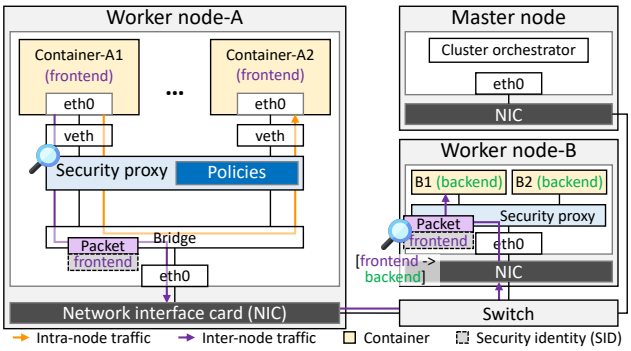


Figure 1: A common container network architecture.

ment integrity and consistency.

In our system, policy enforcement and management tasks for container networking take place within a secure execution environment (a smartNIC) physically separated from the host's OS. Furthermore, *HardWhale* employs hardware-level features [13], [14] to restrict the host's OS from interfering with its operation or manipulating its configuration, significantly reducing attack surfaces exposed to the host side. This isolated design not only provides robust network security but also accelerates traffic inspection tasks for container networking. We implement a full prototype of *HardWhale* on Mellanox Bluefield2 smartNICs [15] and conduct an extensive set of evaluations. The results show that *HardWhale* effectively mitigates network attacks across all layers, even if the host's OS has been compromised. Moreover, it improves TCP latency by up to 5.7x and HTTP latency by up to 2.6x compared to state-of-the-art solutions.

Our contributions are summarized as follows:

- A comprehensive analysis of the limitations of existing container network security solutions in terms of security measures against lateral movement and performance.
- The design and implementation of a novel hardware-based container network security system, *HardWhale*, which leverages a smartNIC to provide physically isolated security policy enforcement/management for containers.
- The comprehensive evaluation of *HardWhale*, demonstrating its practical effectiveness from the security and performance perspectives compared to today's solutions.

II. Background and Motivation

A. Container Networking Architecture

As shown in Fig. 1, containers for microservices are deployed to a container cluster consisting of multiple machines (i.e., nodes). The master node is responsible for scheduling these containers onto one of the worker nodes[1]. On each node, containers have their own isolated network namespace (net-NS) [1], meaning that a container does not have visibility into the network interfaces or traffic of others. To enable communication between containers in different net-NSs, container platforms adopt a bridge network. Packets from a container are first sent to the bridge through a paired virtual interface located on the node side (`veth`), then delivered to the destination container via its `veth` (orange arrows in Fig.1). If the destination container resides on another node, the packet is encapsulated through tunneling protocols (e.g., VXLAN) and then sent over the wire, where it is decapsulated at the receiving node (purple arrows).

[1]Here, we define a node as a worker node unless otherwise noted.

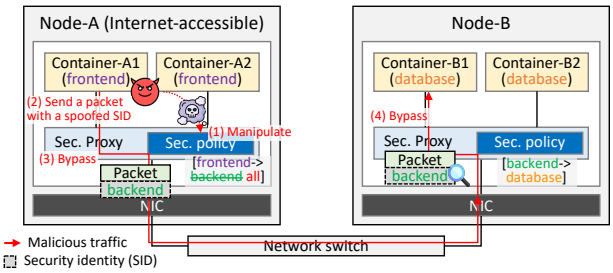
B. Container Network Security Solutions

In container environments, attackers with a malicious container can move laterally within a cluster by compromising other containers through network attacks [5], [6], [16], [17]. This lateral movement involves a sequence of steps, where attackers discover accessible containers using scanning attacks [5]. They then exploit vulnerabilities of the discovered containers by sending malicious packets crafted for specific exploits [17], thereby gaining control over them.

To counteract such threats, today's solutions [5], [8]–[10] enforce network policies, filtering out malicious traffic. Those container network security solutions employ a security proxy for inspecting container traffic, utilizing kernel networking features (e.g., eBPF [18] and iptables) and user space components (e.g., envoy [19]). This proxy intercepts packets at the bridge interface and inspects them based on locally stored L3 to L7 security policies, filtering any out-of-policy packets. Given the ephemeral nature of container IP addresses, which frequently change due to their short lifespan, stable *security identities* (SIDs) are assigned for each container based on their role in microservices (e.g., frontend or backend) and are employed within the context of network policies (e.g., frontends can communicate with backends). For this, the proxy first maps the SRC/DST IP addresses of a packet to corresponding SIDs assigned to the SRC and DST containers. It then compares these SIDs with network policies to determine the authorization between SRC and DST containers. To enforce identity-based policies across a multi-node cluster, the SRC SIDs should be integrated into every packet transmitted between nodes [20]. This integration is vital because the receiving node lacks detailed information of the SRC containers, including their SIDs. During encapsulation, the proxy inserts the SRC SID into the encapsulation header, as shown in Fig. 1. Upon reception, the receiving node extracts the SRC SID and verifies whether the identity is allowed to communicate with containers on it. This ensures that only authorized entities (containers) can initiate communications, thereby mitigating unauthorized lateral movement within the cluster.

C. Threat Model

In this work, we explore container network security by considering two distinct scenarios. The first scenario, known as *container compromise* (C-comp), aligns with previous



(a) An experimental environment: Attackers aim to infiltrate containers on other nodes through network attacks.

```
root@node-A:~# bpftool map dump id 144
key:
ae fd 00 00 00 00 00 01
```
A policy that restricts Container-A1 from communicating with containers on *Node-B*

```
root@container-a1:/# tcpdump tcp port 11211
05:39:24.589318 IP container-a1.53960 > container-b1.11211: Flags [S]
05:39:25.619528 IP container-a1.53960 > container-b1.11211: Flags [S]
05:39:27.635523 IP container-a1.53960 > container-b1.11211: Flags [S]
```

(b) Cilium's policy is stored in Node-A's eBPF map. The attackers fail to initiate communication with containers on Node-B.

```
root@node-A:~# ./policy_disable_script.sh
root@node-A:~# bpftool map dump id 144
key:
00 00 00 00 00 00 00 01
```
An attacker-injected policy that allows all communication of Container-A1 without filtering

```
root@container-a1:/# tcpdump tcp port 11211
05:51:24.332768 IP container-a1.41280 > container-b1.11211: Flags [S]
05:51:24.332944 IP container-b1.11211 > container-a1.41280: Flags [S.]
05:51:24.332968 IP container-a1.41280 > container-b1.11211: Flags [.]
05:51:24.333047 IP container-a1.41280 > container-b1.11211: Flags [P.]
```

(c) The injected policy allows attackers to communicate Node-B's containers and further compromise them.

Figure 2: A real-world attack scenario against Cilium. Attackers can disable Cilium by injecting a poisoned policy.

studies [5], [6], [11]. C-comp occurs when remote attackers compromise Internet-accessible containers by exploiting their vulnerabilities. Here, the attacker's access is confined to the network resources (e.g., traffic) of the compromised container. The second scenario, known as *node compromise* (N-comp), introduces a new perspective on container network security, wherein attackers obtain direct access to the node's OS. Such a compromise often stems from container escape vulnerabilities, which are prevalent in the real world [7], [21]. In terms of network security, attackers can gain full visibility into all container traffic on the node and disrupt container traffic processing. Thus, they no longer have an incentive to conduct network attacks against co-located containers on the same node. Their focus shifts towards *lateral movement*, wherein they conduct network attacks targeting *external containers* on different nodes, with the aim of infiltrating them.

Assumption. In N-comp scenarios, we assume that attackers can access system resources related to containers, including policy files and the bridge network. Our goal is to ensure the effectiveness of inter-node policy enforcement even in such scenarios, thereby protecting external containers from the attackers' lateral movement. Note that the protection of co-located containers from attackers' system-level access (e.g., memory access) has been addressed by other studies [22]–[24] and falls outside our scope (see Section VII). In addition, unlike worker nodes, the master node typically does not operate any containers for microservices, remaining isolated from workloads and external exposure. Communication between the master node and worker nodes is also authenticated and authorized by pre-shared certificates between them [25]. Therefore, we assume that the master node is trustworthy and exclude scenarios in which attackers obtain control over the cluster by exploiting the master node.

D. Limitations of Existing Solutions

While today's solutions are designed to prevent lateral movement through network attacks, they still have serious limitations, including security and performance dimensions.

Lack of isolation for policy enforcement stack. Existing solutions operate under the assumption that nodes in the cluster are trustworthy, deploying their network security measure within each node's OS. While this assumption simplifies the design, it introduces a critical limitation. In N-comp scenarios, the proxy and security policies within the compromised node become exposed to attackers. This exposure presents a severe issue, as the attackers can exploit it in various ways. For instance, they can replace the security proxy with a malicious one without any filtering logic. Furthermore, the security policies can be manipulated by attackers. These attacks disable traffic filtering applied to the node, allowing the attackers to transmit malicious packets to any containers on different nodes. To show this limitation, consider the example depicted in Fig. 2a. Here, containers on Node-A, those accessible from the Internet, are prevented from initiating communication with containers on Node-B, which houses sensitive user data (e.g., bank accounts). Cilium [8] limits such communication by enforcing security policies on each node. In this context, we assume that Container-A1 on Node-A is compromised by attackers (C-comp). The goal of attackers is to move laterally to containers on Node-B through network attacks. However, such an attempt is blocked by Cilium, as shown in Fig. 2b.

Once attackers have escaped from Container-A1 and gained access to Node-A (N-comp), the situation takes a dire turn. With this level of access, they can manipulate the exposed Cilium policies, injecting malicious policies that effectively nullify policy enforcement (step 1 in Fig. 2a). As depicted in Fig. 2c, the injected policy permits all traffic sent by Container-A1. Subsequently, the attackers can send malicious packets employing spoofed SIDs (e.g., backend) that are allowed to communicating with containers on Node-B (step 2). Node-A's Cilium, which is already disabled, fails to filter these packets (step 3). Node-B's Cilium, still operational, permits these packets due to the spoofed SIDs (step 4). As a result, the attackers can communicate with containers on Node-B and further compromise them while bypassing policy enforcement.

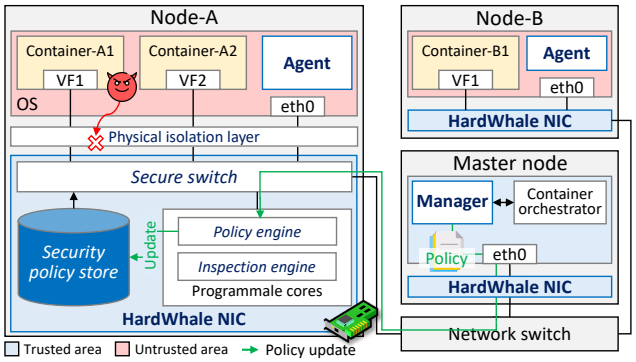


Figure 3: The overall architecture of *HardWhale*.

Limited policy enforcement performance. As containers consisting microservices communicate over L7 protocols (e.g., HTTP REST APIs) [26], it is crucial to restrict L7 connectivity between them to achieve network security. However, existing solutions incur a significant overhead when enforcing such policies for the following reasons. First, the proxy-based inspection mechanism introduces a prolonged and inefficient inspection path for L7 traffic. For example, as the kernel space (e.g., eBPF [18]) is not suitable for payload inspection, every container packet needs to be forwarded to user space components (e.g., envoy [19]). This results in unnecessary packet copying and network stack traversals during packet transmission and reception. Secondly, L7 policy enforcement includes CPU-intensive payload inspection and pattern-matching tasks, placing a substantial load on system resources when applied to numerous containers. We measure L7 policy enforcement overhead by deploying web server and client containers on two nodes with Cilium enabled. We then compared these results to a baseline (Kubernetes without traffic filtering). The outcomes, shown in Fig. 9 (a) in Section VI-A, highlight a two-fold increase in average latency along with a significant increase in tail latency when policy enforcement is active. Also, the server-side node experiences a 73% spike in CPU usage, attributable to the inefficient inspection path and CPU-intensive tasks, as shown in Fig 9 (b).

III. *HardWhale* Design

Our system aims to ensure the effectiveness of network policy enforcement in inter-node container communication, even under N-comp scenarios, while simultaneously accelerating traffic inspection for both intra- and inter-node communication. To accomplish this, *HardWhale* adopts a hardware-based approach by offloading the container network security stack onto smartNICs, which are widely available commodity hardware devices in cloud infrastructures [27]. SmartNICs are equipped with dedicated processors, memory, and accelerators tailored for offloading high-performance networking tasks. Furthermore, they feature a hardware security mechanism referred to as the restricted mode [13], preventing the host from manipulating the internal configuration of these NICs. Our hypothesis is that, given their capabilities, smartNICs are ideally suited to overcome the limitations stemming from the reliance on the node's OS.

A. Overview

The *HardWhale* system, shown in Fig. 3, comprises two main components: the per-node network security stack, residing on each node's smartNIC (referred to as *HardWhale* NIC, HNIC), and a centralized manager. The sequence of its operation unfolds as follows: Upon the boot-up of a node, the HNIC activates the smartNIC's restricted mode, creating a physical isolation layer between itself and the node's OS. This layer protects the HNIC from any potential security breaches emanating from the OS. When containers are deployed on the node, the HNIC controls all container traffic that either originates on the node or arrives from the wire by utilizing its two hardware engines. The secure switch enforces identity-based policies to discard packets that violate L3/L4 policies or contain spoofed identities (e.g., SIDs and IPs). Concurrently, the inspection engine examines packet payloads to enforce L7 policies and to identify malicious content spread across multiple packets. This isolation of policy enforcement protects traffic inspection tasks from attackers and provides accelerated performance, surpassing what software solutions can achieve.

In addition, *HardWhale* adopts a distinctive approach to policy maintenance, securely isolating security policies within the HNIC. The policy engine undertakes real-time policy updates whenever container configurations of the node undergo modifications. This process occurs via an isolated update mechanism and entails direct communication with the manager, situated on a trusted master node (green arrows in Fig. 3). All communications related to updates are protected by secure protocols, such as TLS, and these updates occur within the HNIC, remaining entirely divorced from interactions with the node. This isolated policy management thwarts unauthorized access to the policies from the node, ensuring their security even in N-comp scenarios. In terms of compatibility, our smartNIC-based design can be integrated with existing container platforms via standard interfaces provided them [28], eliminating the need for modifications to container platforms and containerized applications. Additionally, our design can be applied to commodity smartNICs already deployed in cloud infrastructures [27], facilitating various deployment scenarios.

Security implications. Our smartNIC-based design provides protection against security implications originating from the node side. This protection is based on the inherent characteristics of smartNICs. (i) Physical isolation: Software solutions run on the same execution environment with attackers, thereby exposing their control flow to the attackers. Conversely, the HNIC runs on the smartNIC's processor and memory, alongside its embedded OS, ensuring a physically

isolated execution environment from the node's OS. Thus, attackers cannot access the HNIC's control flow and remain unaware of its state. Moreover, the security of smartNICs is further reinforced by a secure boot [14], wherein the integrity of their software stack, including the embedded OS and offloaded programs, is verified during initialization. By transitioning to restricted mode following a secure boot, *HardWhale* ensures the continuous integrity of this isolated environment from its boot stage right through to runtime.

(ii) Limited attack surface: In N-comp scenarios, all system resources tied to software solutions, including file systems, are potential attack surfaces. However, the HNIC, by virtue of its physical isolation, does not depend on the node's system resources for its operations. The activation of the restricted mode [13] further eliminates access to the smartNIC's control interfaces from the node side, countering malicious actions such as attempts to reset offloaded programs or configurations. This protection is also implemented through hardware units on the smartNIC rather than relying on a device driver within the node. As a result, the attack surface is significantly narrowed, removing any direct node-side access to the smartNIC or its offloaded programs.

B. HardWhale Agent and Manager

***HardWhale* agent.** The *HardWhale* agent is a software component that runs on a node's OS and supports the operation of the HNIC installed on the same node. While *inter-node* container traffic relies on the smartNIC for physical transmission, *intra-node* container traffic processing does not involve a smartNIC. This means that the HNIC residing on the smartNIC has no visibility on intra-node container traffic and cannot accelerate the inspection of such traffic. Thus, our system needs to route intra-node traffic to the smartNIC. One strategy is to forward this traffic to the HNIC through the bridge network. Unfortunately, this introduces a complex forwarding path, incurring net-NS traversals and unnecessary packet copying, leading to increased latency. To overcome these challenges, the agent establishes a direct link between each container and the HNIC, utilizing Single Root I/O Virtualization (SR-IOV) [29]. This link enables direct packet exchange with the HNIC while bypassing the bridge network, facilitating thorough inspection of both inter- and intra-node container traffic by the HNIC.

The agent functions as an integrated plugin for container network interfaces (CNIs) [28], seamlessly aligning with existing container platforms. Once the initial configuration for a new container is completed (e.g., establishment of a net-NS), container platforms invoke the agent. Upon invocation, the agent replaces the container's veth interface with an SR-IOV virtual function (VF), which is connected to the HNIC. Throughout the container's lifecycle, both its intra- and inter-node communication processes benefit from efficient acceleration and security measures provided by the HNIC. The VF also performs a spoofing prevention check for *intra-node* communication. Whenever the container sends a packet, the VF verifies whether the packet's source address matches the assigned address of the container, dropping any spoofed packets before forwarding them to the HNIC.

Security implications of agents. While the HNIC operates within a physically isolated layer, the agent resides on the node's OS. Consequently, in N-comp scenarios, attackers may attempt to exploit the agent to bypass our policy enforcement. For instance, attackers can remove VFs within containers, thereby disrupting the HNIC from receiving intra-node container traffic. However, it is crucial to emphasize that security implications arising from a compromised agent do not compromise *HardWhale*'s ability to block lateral movement in N-comp scenarios. Regardless of the status of VFs, inter-node traffic originating from the node is routed to the smartNIC and subjected to comprehensive inspection by the HNIC. As a result, out-of-policy packets targeting *external containers* are prevented from being transmitted from the compromised node to another node. Furthermore, the *policies* for this inspection remain inaccessible to attackers due to the physical isolation.

***HardWhale* manager.** The *HardWhale* manager plays a pivotal role in our system and is responsible for updating security policies on each HNIC. Situated on a trusted master node, it adopts an event-driven approach (e.g., Kubernetes API [30]) to collect container-related events for policy updates. The policy update process unfolds as follows: (1) When an event relevant to the current policy set occurs, such as container creation, the manager generates an updated policy. (2) The manager compiles this policy into a specialized format (e.g., match-action tables) and securely sends it to the target HNIC, which requires an update due to the event. (3) Communication regarding policy updates is encrypted and authenticated using pre-shared keys between the manager and the target HNIC, employing TLS. This ensures data integrity and confidentiality during the transmission. (4) The HNIC decrypts the received policy at the data plane, applying it to the security policy store.

To efficiently manage policies for containers distributed across multiple nodes, the manager adopts a strategy of deploying policies on a per-node basis. Therefore, an HNIC installed on a specific node maintains only policies related to containers running on that node, resulting in updates to the policies of an HNIC only when relevant events are detected. This enables efficient policy updates and minimizes disruption to microservices during the update process.

C. Secure Switch

Here, we examine the three components of the HNIC individually. As shown in Fig. 4, the secure switch is the core of the HNIC, implementing identity-aware L3/L4 policy enforcement through the following three pipeline states.

(i) Pre-filtering. This pipeline stage performs initial tasks, including header parsing and connection tracking, and it pre-

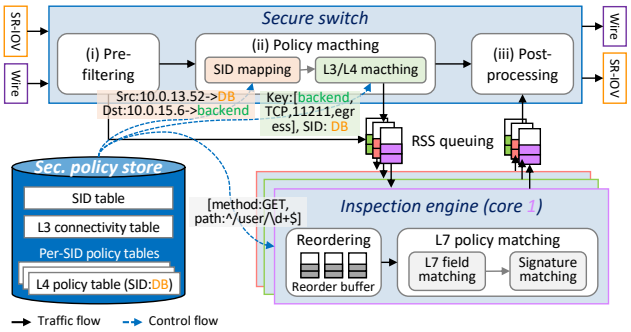


Figure 4: The *HardWhale* NIC's packet processing pipeline.

filters packets that are irrelevant to the enforcement process. The secure switch receives packets from two sources: the network wire and the SR-IOV data path, originating from local containers. For packets arriving from the network wire (*ingress packets*), a decapsulation process is applied. Meanwhile, for packets originating from the SR-IOV data path (*egress packets*), the secure switch conducts an additional node-level spoofing check. This check complements the container-level spoofing checks performed by VFs on each container. Each node in the cluster can only allocate addresses within a dedicated address range assigned by the master node to containers residing on that node. The secure switch discards egress packets with invalid source addresses that fall outside the node's assigned range, preventing spoofed packets from being transmitted to other nodes. Also, this check operates within the HNIC, ensuring its resilience under N-comp cases.

The secure switch maintains a record of L4 connections involving containers on its node, facilitating stateful and efficient packet inspection. As shown in Fig. 4, for packets within already established connections, L3/L4 policy enforcement is skipped, with packets sent to the inspection engine along with their connection records for L7 policy enforcement. This stateful processing reduces redundant packet inspection, enhancing pipeline processing throughput.

(ii) Policy matching. The secure switch enforces L3/L4 policies through a set of match-action tables. This process commences by mapping a packet's IP addresses to the SIDs of their source and destination containers (*SSID* and *DSID*) via the SID table. For instance, as shown in Fig. 4, the SRC and DST IP addresses of the egress packet are mapped to *DB* and *backend* SIDs, respectively. For ingress packets, SSIDs are derived directly from their encapsulation header. During this mapping, both egress and ingress packets failing to resolve their SSIDs or egress packets already having encapsulated SSIDs are discarded to thwart SID spoofing attacks. After that, the switch verifies the authorization of SSID and DSID based on the L3 connectivity table, blocking unauthorized traffic between containers.

To enforce L4 policies, a distributed table structure is employed, with distinct policy tables for each SID. Initially, a matching key is created based on the packet's DSID, protocol, DST port, and traffic direction. This key is then used to query the relevant policy from the policy table associated with the SSID. As exemplified by the egress packet processing in Fig. 4, the policy lookup is placed in the policy table associated with the *DB* SID. For ingress packets, the SSID and SRC port are used for the matching key instead of the DSID and destination port, and policy lookup also occurs in the table associated with the DSID. Any packets violating the retrieved policy are not forwarded to subsequent pipeline stages. This per-SID policy partitioning minimizes performance impact during enforcement compared to node-global (e.g., iptables).

(iii) Post-processing. After policy matching, the secure switch updates the L4 connection records for legitimate packets and subsequently directs them to their destinations. Intra-node packets are routed based on the hardware port associated with the destination container's VF. For inter-node packets, an encapsulation process is employed before their transmission. Their SSIDs are embedded in the encapsulation headers for policy matching at the receiving node.

D. Inspection Engine

Our inspection engine is designed to accelerate L7 policy enforcement by utilizing hardware accelerators within the smartNIC. In the lower section of Fig. 4, the inspection engine comprises multiple instances, each allocated to a dedicated smartNIC core, enabling parallel packet processing. The secure switch directs packets, along with their connection records, to a specific instance through RSS (Receive-Side Scaling) queuing. RSS ensures that packets belonging to the same connection are routed to the same engine instance, enhancing cache utilization on the smartNIC. Upon receiving packets, this engine initially assesses if they are out-of-order (OoO) packets. OoO packets are placed into reorder buffers and sequenced correctly to detect malicious patterns across multiple packets. Subsequently, the inspection engine retrieves L7 policies for the received packets. L7 policies encompass protocol-specific fields, such as HTTP method and path, along with general detection signatures. Now, packets are subjected to regex accelerators that are integrated within the smartNIC. These accelerators execute regular expression matching against packet payloads at a line rate, thereby governing L7 connectivity between containers based on policies. For instance, frontends may be restricted to exclusively access the path "GET /user/d{3}" of backends. While existing solutions primarily inspect known L7 headers (e.g., HTTP and DNS), our engine goes further, effectively scrutinizing malicious content deep inside payloads. When L7 policies contain detection signatures, the engine examines packet payloads from specified offsets, discarding packets containing the designated signatures. This deeper inspection blocks exploit packets targeting known vulnerabilities of containers (e.g., Log4j). After that, benign

packets are returned to the secure switch's final stage.

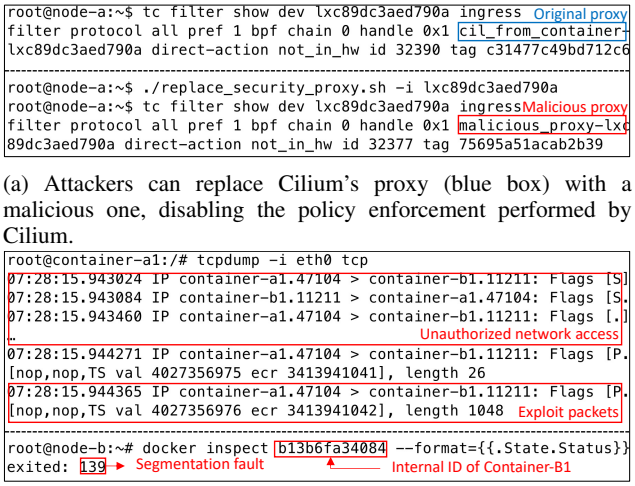
E. Policy Engine

The policy engine is responsible for real-time security policy updates. As we already discussed in Section III-B, whenever a container event occurs, the manager on the master node analyzes it, determining the necessity for a corresponding policy update. If an update is required, the manager compiles updated policies for the specific event and dispatches these policies to the HNICs that necessitate these updates. The manager initiates this communication through control packet exchanges. All control packets are protected through secure protocols (e.g., TLS), which can be offloaded to smartNICs. This secure communication blocks potential threats targeting control packets (e.g., eavesdropping or MITM) during their transmission. On the recipient side, the policy engine, which runs on smartNIC cores, (1) authenticates and decrypts the control packet. Any packets that fail this step are discarded, ensuring the security of policy updates. (2) After that, the policy engine extracts the compiled policies from these packets, creating new policy table entries. (3) The created entries are updated to the policy store through our concurrent update mechanism.

Our policy engine is designed to support concurrent policy updates, implementing agile policy deployments during runtime. Each policy engine instance runs on a dedicated smartNIC core, each furnished with its own update queue when accessing table entries. The table entries for updates are scheduled to one of these instances based on their respective table name and content, ensuring that a given entry consistently interfaces with the same instance. This design eliminates the necessity for locking data structures for table entries and facilitating concurrent updates through the utilization of multiple per-core update queues. Note that entry updates can be performed in runtime without blocking pipeline processing. Policy update operations take place within the HNIC without involving the node's OS. Control packets are protected by TLS, and they are not directed to the node's OS after updates. These designs guarantee the isolation of policy management in N-comp scenarios.

IV. Implementation

We implemented a full prototype of *HardWhale* using Mellanox Bluefield2 smartNIC [15]. The *HardWhale* NIC and its three hardware engines were implemented using 4K lines of C code with the Mellanox API [31]. The smartNIC used for the prototype consists of programmable ARM cores and various hardware accelerators, which allow the security modules to run on the ARM cores and make use of the accelerators. To ensure full isolation between the node's OS and the smartNIC, we configured the smartNIC in restricted mode [31], which prevents the OS from accessing it. The agent and manager were implemented using 1.5K lines of Python code with Kubernetes APIs [30]. A control packet



(a) Attackers can replace Cilium's proxy (blue box) with a malicious one, disabling the policy enforcement performed by Cilium.

(b) Attackers can now access Container-B1 and compromise it by sending exploit packets targeting its known vulnerability [32].

Figure 5: In N-comp scenarios, even state-of-the-art solutions are unable to prevent lateral movement.

between the manager and the *HardWhale* NIC is protected by TLS, which can be offloaded to modern smartNICs. The intra-node spoofing check function is implemented with TC eBPF programs [18] and is attached to each VF.

V. Security Evaluation

Here, we show the effectiveness of *HardWhale* from the security perspective by answering the two questions:

- **Q1**: Can *HardWhale* protect external containers from attackers within a compromised node in N-comp scenarios?
- **Q2**: Can *HardWhale* securely maintain its policies and update them on-the-fly under N-comp scenarios?

For brevity, we omit security evaluations for C-comp scenarios, which have been extensively examined in previous studies [5], [6]. While our focus on this evaluation is N-comp scenarios, *HardWhale* can protect co-located/external containers and policies in C-comp scenarios as well.

A. Evaluation Environment

For security evaluations, we employ the same environment shown in Fig. 2a. In this setup, attackers on Node-A attempts to breach Node-B using network attacks. All nodes run Kubernetes v1.27.3 on Ubuntu 22.04 and utilize Cilium for secure container networking. Also, Bluefield2 smartNICs with *HardWhale* prototypes are deployed on each node.

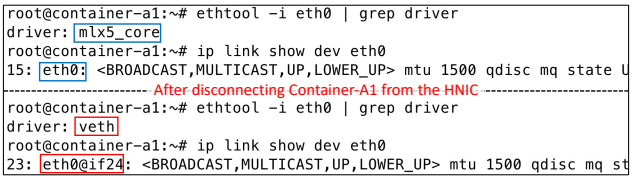
B. Isolation of policy enforcement

Without *HardWhale*. To address our first question, **Q1**, we have conducted evaluations to underscore the effectiveness of our policy enforcement isolation. In N-comp scenarios, attackers possess the capability to compromise *external containers* through network attacks, even in the presence of Cilium. This vulnerability arises because the attackers can

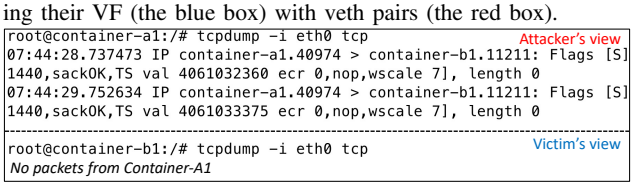
nullify Cilium's policy enforcement. This malicious action, as depicted in Fig. 5a, involves replacing Cilium's security proxy with a malicious one that refrains from conducting any security checks. Consequently, the attackers can initiate communication with Container-B1 on Node-B, a scenario that should be restricted, as shown in the upper part of Fig. 5b. Moreover, the attackers may employ SID spoofing attacks to bypass the receiver-side filtering on Node-B. This failure in policy enforcement leads to further attacks on victim containers. As shown by the second red box in Fig. 5b, the attackers can exploit vulnerabilities within Container-B1 by sending *exploit packets* designed for buffer overflow [32]. The result is a segmentation fault within the victim container, representing the initial step in an exploit chain, as shown in Fig. 5b.

With *HardWhale*. In stark contrast, *HardWhale* prevents such attacks by implementing policy enforcement isolation under N-comp scenarios. Even in the face of potential threats, such as attackers disconnect Container-A1 from the HNIC by replacing its VF to the *veth* pair (shown in Fig. 6a), the security provided by the HNIC remains robust in blocking network attacks targeting external containers. As shown in Fig. 6b, the attackers' attempt to communicate with Container-B1 from A1 (or any containers on Node-A) is thwarted by the HNIC. In addition to the above L3/L4 policy enforcement, *HardWhale* extends its security scope to L7 communication. To exemplify this, we intentionally permit Container-A1 to communicate with B1 through TCP port 11211. During this communication, the attackers (A1) attempt to exploit the victim (B1)'s vulnerability [32] by transmitting *exploit packets* containing malicious contents, as depicted by the red box in Fig. 6c. *HardWhale* mitigates such attacks by conducting a comprehensive payload inspection. It discards packets containing exploit payloads while allowing benign ones (blue boxes in Fig. 6c).

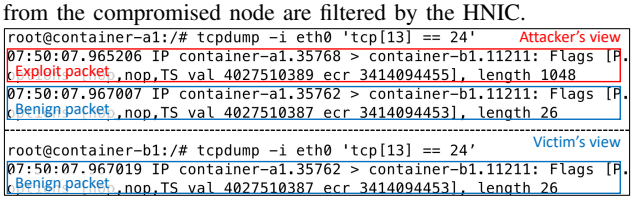
Spoofing prevention. In N-comp scenarios, existing solutions cannot restrict attackers from initiating spoofing attacks, which are initial steps in more extensive attacks. For instance, consider the scenario in which the attackers (A1) attempt to transmit a spoofed packet with the IP address of Container-B2 or any other containers that are allowed to communicate with -B1 (indicated by the red box in Fig. 7). Furthermore, they may craft a malicious packet with a spoofed source SID (corresponding to Container-B2) within its encapsulation header to circumvent receiver-side policy enforcement (the purple box in Fig. 7). However, such malicious attempts are thwarted by the HNIC. This is because all inter-node packets originating from the compromised node undergo the isolated HNIC's inspection before their transmission. Packets with spoofed IPs are discarded during the pre-filtering stage, and those containing spoofed source SIDs are blocked at the SID mapping stage (see Section III-C). As shown in the bottom of Fig. 7, only packets bearing valid IP addresses and SIDs are allowed



(a) Attackers can disconnect containers from the HNIC by replacing their VF (the blue box) with veth pairs (the red box).



(b) Regardless of VFs, malicious inter-node packets coming out from the compromised node are filtered by the HNIC.



(c) The HNIC can detect and discard exploit packets targeting the victim's vulnerability [32] while allowing other benign packets.

Figure 6: The effectiveness of the *ourtool* NIC's isolated policy enforcement under N-comp scenarios.

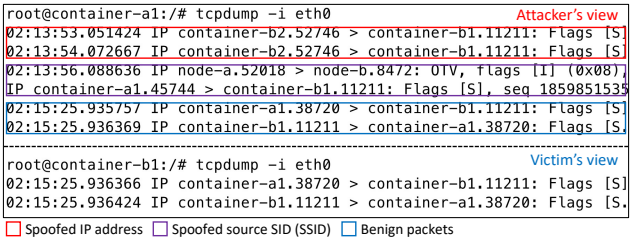


Figure 7: The effectiveness of *HardWhale* against IP and SID spoofing attacks.

to continue their journey to Container-B1. Note that we intentionally permitted Container-A1 to communicate with B1 in this scenario to show the outcomes.

C. Isolation of policy management

To address the second question, **Q2**, we delve into the effectiveness of *HardWhale*'s hardware-isolated policy management. Existing solutions store security policies within a node's OS under the assumption that the node remains trustworthy. For instance, Cilium and Bastion [5] rely on eBPF-based key-value stores, while Calico [9] employs a set of iptables rules to manage these policies. As discussed in Section II-D, this approach exposes policies to potential attacks in N-comp scenarios, allowing attackers to manipulate exposed policies (see Fig. 2c). In contrast, *HardWhale* securely maintains its policies within the HNIC, thereby isolating them from potential attacks originating from the

```
root@hardwhale-nic-node-a:/# tcpdump -i p0 tcp port 443
16:31:52.585599 IP hw-manager.35014 > node-a.https: Flags [S], seq 42
16:31:52.585697 IP node-a.https > hw-manager.35014: Flags [S.], seq 3
16:31:52.585885 IP hw-manager.35014 > node-a.https: Flags [.], ack 1,
…                                   Secure policy updates over TLS
16:31:52.594317 IP hw-manager.35014 > node-a.https: Flags [P.], seq 5
16:31:52.594428 IP node-a.https > hw-manager.35014: Flags [P.], seq 1
16:31:52.607850 IP node-a.https > hw-manager.35014: Flags [FP.], seq
```

(a) Packets visible from the HNIC on Node-A: Control packets are protected through TLS. The policy engine communicates with the *hw-manager* using Node-A's IP address and TCP 443 port.

```
root@hardwhale-nic-a:/# tcpdump -i p0 tcp port 443
16:32:39.570114 IP hw-manager.45746 > node-a.https: Flags [S], seq 39
16:32:39.570222 IP node-a.https > hw-manager.45746: Flags [S.], seq 2
16:32:39.570395 IP hw-manager.45746 > node-a.https: Flags [.], ack 1,
…                   Malicious policy update attempts using invalid secrets
16:32:39.576966 IP hw-manager.45746 > node-a.https: Flags [.], ack 14
16:32:39.578114 IP hw-manager.45746 > node-a.https: Flags [P.], seq 5
16:32:39.578162 IP hw-manager.45746 > node-a.https: Flags [R.], seq 5
```

(b) The policy engine rejects malicious control packets with invalid secrets, ensuring the integrity of policy updates.

Packets visible within the HNIC

```
20:13:22.995209 IP node-b.38600 > node-a.8472: OTV, flags [I] (0x08),
IP 10.0.1.158.http > 10.0.0.137.34360: Flags [F.], seq 854, ack 76, w
20:13:22.995285 IP node-a.46325 > node-b.8472: OTV, flags [I] (0x08),
IP 10.0.0.137.34360 > 10.0.1.158.http: Flags [.], ack 855, win 502, o
20:13:22.995527 IP node-a.https > hw-manager.43782: Flags [P.], seq 1
20:13:22.995639 IP hw-manager.43782 > node-a.https: Flags [.], ack 14
```

Control packets

Packets visible from the node's operating system

```
20:13:22.998136 IP 10.0.1.158.80 > 10.0.0.137.34360: Flags [F.], seq
20:13:22.998306 IP 10.0.0.137.34360 > 10.0.1.158.80: Flags [.], ack 8
20:13:23.532422 IP 10.0.0.137.34370 > 10.0.1.158.80: Flags [S], seq 4
20:13:23.532478 IP 10.0.1.158.80 > 10.0.0.137.34370: Flags [S.], seq
```

(c) After policy updates, the policy engine does not forward control packets to the node's OS while routing other container traffic.

Figure 8: The effectiveness of policy management isolation.

compromised node. Moreover, the HNIC possesses the capability to securely update these isolated policies through communication with the remote manager. Fig. 8a shows this process where control packets for updates are both encrypted and authenticated through TLS. The policy engine of the HNIC exclusively applies updates from valid control packets, promptly discarding those that fail verification, as depicted in Fig. 8b. This ensures the integrity and confidentiality of policy updates, safeguarding control packets from potential attacks during transmission. Once policies are updated, the HNIC does not forward the used control packets to the network stack of the node, as shown in Fig. 8c. Instead, it forwards only typical container packets, ensuring that the attackers remain oblivious to policy updates.

VI. Performance Evaluation

Here, we evaluate the performance enhancement achieved by *HardWhale* compared to existing security solutions for container networking. For this, we set up a cluster consisting of two physical nodes equipped with an Intel Xeon 4114 CPU and 64GB of RAM. These nodes have the *HardWhale* prototype and are connected via a 40GbE cable.

A. End to End Security Policy Enforcement Performance

We first measure the performance of *HardWhale*'s policy enforcement and compare it with Cilium, the fastest of

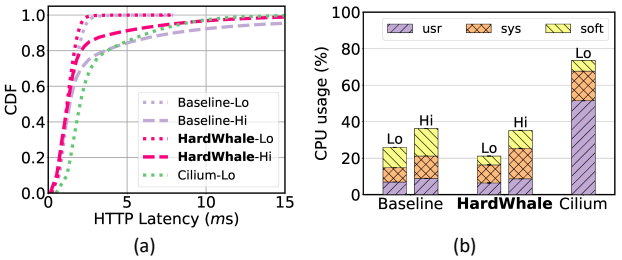


Figure 9: E2E policy enforcement performance. The CPU usage is divided into three types: user-space (`usr`), kernel-space (`sys`), and software interrupt handling (`soft`).

the existing solutions. We deploy an Nginx server and client containers on each node and apply security policies that restrict HTTP communication between them based on specific TCP ports and HTTP paths. To generate traffic, we employ a well-known HTTP benchmark tool, *wrk2*. For a comprehensive analysis, we consider various traffic load scenarios. We configure the benchmark tool to generate 30K and 100K HTTP requests per second (RPS) to simulate low (*Lo*)- and high-traffic (*Hi*) load scenarios. Also, we set up a baseline using Kubernetes without any security checks or offloaded features. Note that we omit the measurement of Cilium in the *Hi* load scenario, as it exceeds Cilium's maximum throughput capacity.

HTTP performance. Fig. 9 (a) shows the CDF of HTTP round-trip latency between the client and server during security policy enforcement under both *Lo* and *Hi* load scenarios. Under the *Lo* scenario, *HardWhale* maintains an approximate 2.3x faster average latency than Cilium. Notably, in terms of tail latency, *HardWhale* achieves a P99 latency of 2.489ms, in stark contrast to Cilium's 12.4ms P99 latency. This result represents a 4.9x improvement and is comparable to the performance of the baseline. Moreover, *HardWhale*'s performance advantages become even more evident under the *Hi* load scenario. While Cilium fails to handle this high traffic load, *HardWhale* attains up to 1.8 times faster average latency than the baseline. Even in this high traffic load, *HardWhale* handles 90% of HTTP packets within 4.32ms while enforcing L3 to L7 security policies. Table I shows the HTTP latency of both systems under different numbers of concurrent clients. While Cilium shows linear growth in average and maximum latency as the number of clients increases, *HardWhale* maintains consistent performance and remains twice as fast as Cilium.

To evaluate the scalability of both systems concerning their support for security policies, we measure each solution's maximum HTTP RPS rate under varying quantities of security policies from one to 500. Given that most HTTP packets used for microservices contain short URLs and are small in size [33], policies used in this evaluation are configured to contain randomized HTTP paths, each with a length of 20. Table II shows the HTTP throughput of these

Table I: Average HTTP latency measurements (2K RPS per client, the value in parentheses is the maximum latency.)

# of clients	*4 clients*	*8 clients*	*12 clients*
HardWhale	1.08 (5.51) ms	1.09 (5.59) ms	1.06 (5.12) ms
Cilium	1.63 (10.1) ms	1.87 (12.1) ms	2.02 (17.6) ms

Table II: HTTP throughput measurement results.

HTTP RPS	*Baseline*	*HardWhale*	*Cilium*
Single policy	117.5K	115.3K	28.5K
100 policies		112.6K	27.8K
500 policies		108.2K	27.6K

systems. *HardWhale* outperforms Cilium more than 3x while maintaining comparable performance with the baseline. In contrast, Cilium experiences about 80% throughput degradation compared to the baseline. Conversely, *HardWhale* records a minor 7% degradation. Both solutions deliver stable performance, regardless of the number of policies, thanks to their efficient multi-pattern matching algorithms.

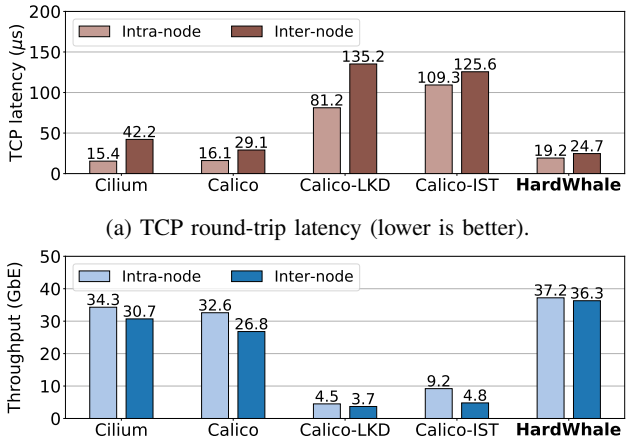
These improvements in latency and throughput achieved by *HardWhale* can be attributed to several key factors. Firstly, Cilium incurs redundant packet copy overhead and network stack traversals due to redirections of HTTP traffic between the kernel (eBPF) and user space components (envoy [19]). Secondly, the CPU-intensive nature of L7 policy enforcement presents challenges for software solutions. *HardWhale* adeptly addresses these limitations by offloading CPU-intensive tasks to the smartNIC. This design also optimizes inter-node communication by offloading the traffic encapsulation task, resulting in more stable and rapid latency than the baseline. Note that we omit the evaluation for intra-node communication for brevity, as the results are similar to those of the above inter-node communication.

CPU usage. Lastly, we analyze CPU resource consumption to assess the overhead associated with security policy enforcement. Using the *mpstat* tool, we monitor CPU usage on the server-side node. Under a 30K RPS HTTP traffic load (*Lo*), Cilium consumes a substantial 73% of CPU resources, with about 50% attributed to its user-level proxy inspecting HTTP payloads, as shown in Fig. 9 (b). In contrast, *HardWhale* exhibits significantly lower CPU usage, achieving a 4-fold reduction compared to Cilium. Moreover, *HardWhale* maintains similar CPU usage rates as the baseline under both *Lo* and *Hi* traffic scenarios. This underscores *HardWhale*'s efficient policy enforcement capability that allows a node to allocate more system resources for microservices.

B. Microbenchmark

To show the overhead introduced by *HardWhale*, we measure TCP round-trip latency and throughput under *inter-node* and *intra-node* communication scenarios.

Intra-node communication. In Fig. 10a, it can be seen that



(a) TCP round-trip latency (lower is better).

(b) TCP throughput (higher is better).

Figure 10: TCP latency and throughput measurement results.

the TCP latency of containers on the same node is slightly increased when using *HardWhale*, with an average of 3.5μs compared to CNIs (Cilium and Calico). This increase can be attributed to the PCIe overhead incurred when transferring intra-node traffic from containers to the smartNIC's memory in *HardWhale*. In contrast, other solutions handle intra-node traffic at the kernel-level using eBPF. However, when combining CNIs with user space proxies (Linkerd [34] and Istio [10]) to enable L7 inspection, *HardWhale* outperforms these solutions by up to 5x. This is because these solutions incur high overhead when redirecting packets between kernel and user space components. In contrast, *HardWhale* processes container traffic using the smartNIC, eliminating such overhead. Similarly, as shown in Fig. 10b, it outperforms other solutions in TCP throughput by up to 3.8x.

Inter-node communication. *HardWhale* shows the superior performance of both TCP latency and throughput for inter-node communication involving packet en(de)capsulation. It achieves up to 5.27x faster TCP latency and 4.95x higher throughput than existing solutions, as shown in Fig. 10a and 10b. These performance gains can be attributed to *HardWhale*'s ability to leverage hardware accelerators for packet processing, including traffic en(de)capsulation. Additionally, the SR-IOV-based direct link between container VFs and the *HardWhale* NIC further enhances the efficiency of processing for inter-node traffic. These optimizations enable *HardWhale* to achieve line-rate performance. In contrast, software solutions incur overhead during the en(de)capsulation processes and cannot achieve the same performance as *HardWhale*.

VII. Related Work

Container network security. Several studies [5], [35] examine the network security landscape of containers and demonstrate that current container networks are vulnerable to network attacks. To mitigate such threats, previous studies

has relied on software-based policy enforcement, including eBPF-based filtering solutions [5], [8], [9] and specialized user space proxies equipped with various functionalities [10], [34]. However, these studies are developed without considering N-comp scenarios. Also, their software-based nature places a heavy burden on system resources when inspecting packet payloads sent by numerous containers.

Hardware-based Network Security. Numerous studies [11], [36]–[39] investigate the offloading of CPU-intensive network security functions to data plane devices within cloud environments. For instance, Teng et al. [38] implement a stateful L3/4 firewall on a P4 switch, which enables efficient inspection of inter-node traffic. However, these studies lack visibility into network attacks within intra-node container communication, as this type of traffic does not involve switch processing. Moreover, they still face limitations in L7 inspection [36], [37]. For example, they support limited types of L7 protocols (e.g., DNS) and fail to detect patterns across out-of-order packets, which can be a significant drawback in container environments. To address such problems, recent studies adopt smartNICs [11], [39], thereby facilitating the protection of inter- and intra-node communication. Unfortunately, they are not designed with N-comp scenarios in mind. For example, they still rely on the node's OS to update policies and lack protection mechanisms to prevent attackers from manipulating their configurations. In contrast, *HardWhale* supports comprehensive L7 inspection capabilities, including regex matching against various L7 protocols with variable-length fields and detection of malicious contents across out-of-order packets. It also ensures the isolation of policy enforcement from the node, guaranteeing its resilience in N-comp scenarios.

Hardware-based System Security. Another line of research focuses on enhancing the weak isolation of containers using hardware enclaves [22]–[24]. SCONE [22], for instance, isolates processes in containers from the OS kernel and ensures data and code confidentiality of containers by utilizing Intel SGX. While these efforts shield the data and code of co-located containers from hostile system-level access in N-comp scenarios, they fall short in protecting *external containers* from lateral movements by attackers. The limitation arises from a constricted focus on the execution of container processes and the protection of their data within enclaves, without consideration of the isolation for container network policy enforcement from the node's OS. Thus, the node's OS retains control over container traffic, allowing attackers to compromise external containers by sending malicious traffic. While enclave-based solutions are orthogonal to our paper, integrating them with *HardWhale* can forge a more holistic security framework, capable of shielding containers from system- and network-level threats under N-comp scenarios.

Some customized virtualization systems (e.g., Nitro [40]) offload key virtualization functionalities into hardware, reducing the attack surfaces exposed to attackers. For instance, they can offload all I/O operations to PCIe cards from the hypervisor. While these studies may support robust network security in N-comp scenarios, they require extensive modifications across all layers of container systems. This includes the need for customized hardware (processors and PCIe cards) and redesigned hypervisors. These issues limit their applicability to specific vendors (e.g., AWS), preventing a general use in cloud environments. In contrast, *HardWhale* adopts a smartNIC-based design that leverages off-the-shelf hardware devices commonly found in cloud environments. This design does not require modifications to upper-layer software and can be deployed in general cloud environments.

VIII. Limitations and Discussions.

Limited resources in a smartNIC. One limitation of *HardWhale* is the amount of memory available in the smartNIC. The number of policies and active TCP flows that can be maintained at line-rate is limited by the smartNIC's memory. In particular, *HardWhale* can track around 1M flows and enforce a maximum of 8K policies per-node. For a larger number of policies, external memory access is necessary. The scalability in supporting a large number of containers is also a limitation of our current design. To inspect intra-node traffic, *HardWhale* should connect all containers on the node to the HNIC using SR-IOV. This means that the number of containers the HNIC can support on a single node is limited by the number of SR-IOV VFs supported by the smartNIC. However, it is important to consider how container platforms manage containers. Container platforms group multiple containers into a single pod, with resource allocations (e.g., net-NSs) being managed at the pod level. Furthermore, commodity smartNICs support up to 254 VFs, which is sufficient to support all pods running on a single node in a general cluster configuration [41].

PCIe bus overhead. *HardWhale* employs a smartNIC to inspect all container traffic originating from a node. This design has a drawback in intra-node container communication scenarios as the all traffic needs to be routed to the smartNIC through the PCI bus, incurring extra overhead. In our evaluation, we observed that the traversal of the PCIe datapath resulted in an average latency overhead of $3.5\mu s$ for TCP communications. Nonetheless, this overhead is negligible compared to the robust security benefits provided by *HardWhale* under N-comp scenarios.

IX. Conclusion

In this paper, we present *HardWhale*, a hardware-based network security system for containers that employs a smartNIC to inspect container traffic and securely maintain network policies, independent of the node's OS. Our evaluations demonstrate that *HardWhale* effectively prevents various network attacks even in scenarios where the OS is compromised, and it enhances network latency, achieving up to a 5.7-fold improvement over existing solutions.

ACKNOWLEDGMENT

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2024-RS-2023-00258649) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).

REFERENCES

[1] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud Container Technologies: a State-of-the-art Review," *IEEE Transactions on Cloud Computing*, vol. 7, no. 3, pp. 677–692, 2017.

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

[3] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated System Call Policy Generation for Container Attack Surface Reduction," in *Proceedings of International Symposium on Research in Attacks, Intrusions and Defenses*, 2020.

[4] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, "SPEAKER: Split-phase Execution of Application Containers," in *Proceedings of International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 230–251.

[5] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin, "BASTION: A Security Enforcement Network Stack for Container Networks," in *Proceedings of USENIX Annual Technical Conference*, 2020, pp. 81–95.

[6] S. Sultan, I. Ahmad, and T. Dimitriou, "Container Security: Issues, Challenges, and the Road Ahead," *IEEE Access*, vol. 7, 2019.

[7] "Kubernetes Privilege Escalation," https://i.blackhat.com/USA-22/Thursday/US-22-Avrahami-Kubernetes-Privilege-Escalation-Container-Escape-Cluster-Admin.pdf, 2022.

[8] "Cilium CNI," https://www.cilium.io/, 2023.

[9] "Project Calico," https://www.projectcalico.org/, 2023.

[10] "The Istio service mesh," https://istio.io/, 2023.

[11] M. You, J. Nam, M. Seo, and S. Shin, "HELIOS: Hardware-assisted High-performance Security Extension for Cloud Networking," in *Proceedings of the ACM Symposium on Cloud Computing*, 2023, pp. 486–501.

[12] Z. Jian and L. Chen, "A Defense Method against Docker Escape Attack," in *Proceedings of the International Conference on Cryptography, Security and Privacy*, 2017, p. 142–146.

[13] "NVIDIA BlueField DPU Zero trust mode," https://docs.nvidia.com/doca/sdk/modes-of-operation/index.html, 2023.

[14] "Secure boot for dpu," https://docs.nvidia.com/networking/display/bluefielddpubspv422/uefi+secure+boot, 2024.

[15] "Nvidia Bluefield-2 DPU," https://www.nvidia.com/en-us/networking/products/data-processing-unit/, 2023.

[16] F. Minna, A. Blaise, F. Rebecchi, B. Chandrasekaran, and F. Massacci, "Understanding the Security Implications of Kubernetes Networking," *IEEE Security & Privacy*, 2021.

[17] Y. Yang, W. Shen, B. Ruan, W. Liu, and K. Ren, "Security Challenges in the Container Cloud," in *Proceedings of IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications*, 2021, pp. 137–145.

[18] "eBPF Introduction, Tutorials," https://ebpf.io/, 2023.

[19] "Envoy proxy," https://www.envoyproxy.io/, 2023.

[20] "Security Identieis," https://docs.cilium.io/en/latest/security/network/policyenforcement/#policy-enforcement/, 2023.

[21] M. Reeves, D. J. Tian, A. Bianchi, and Z. B. Celik, "Towards Improving Container Security by Preventing Runtime Escapes," in *Proceedings of IEEE Secure Development Conference*, 2021, pp. 38–46.

[22] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell *et al.*, "{SCONE}: Secure linux containers with intel {SGX}," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 689–703.

[23] Y. Jia, S. Liu, W. Wang, Y. Chen, Z. Zhai, S. Yan, and Z. He, "{HyperEnclave}: An open and cross-platform trusted execution environment," in *2022 USENIX Annual Technical Conference*, 2022, pp. 437–454.

[24] M. Li, Y. Xia, and H. Chen, "Confidential serverless made efficient with plug-in enclaves," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*, 2021, pp. 306–318.

[25] "Controlling Access to the Kubernetes API," https://kubernetes.io/docs/concepts/security/controlling-access/, 2024.

[26] M. Grambow, L. Meusel, E. Wittern, and D. Bermbach, "Benchmarking Microservice Performance: a Pattern-based Approach," in *Proceedings of the Annual ACM Symposium on Applied Computing*, 2020, pp. 232–241.

[27] "Heavy Reading SmartNIC Survey," https://img.lightreading.com/downloads/Heavy-Reading-SmartNIC-Survey-2021.pdf, 2021.

[28] "Container Network Interface," https://www.cni.dev/, 2023.

[29] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High Performance Network Virtualization with SR-IOV," *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, 2012.

[30] "K8s APIs," https://kubernetes.io/docs/reference/, 2023.

[31] "DOCA SDK," https://docs.nvidia.com/doca/sdk/, 2023.

[32] "CVE-2016-8704," https://nvd.nist.gov/vuln/detail/CVE-2016-8704/, 2016.

[33] F. Romero, G. I. Chaudhry, Í. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "FaaT: A Transparent Auto-scaling Cache for Serverless Applications," in *Proceedings of the ACM symposium on cloud computing*, 2021, pp. 122–137.

[34] "The Linked service mesh," https://linkerd.io/, 2023.

[35] G. Budigiri, C. Baumann, J. T. Mühlberg, E. Truyen, and W. Joosen, "Network Policies in Kubernetes: Performance Evaluation and Security Analysis," in *Proceedings of Joint European Conference on Networks and Communications & 6G Summit*, 2021, pp. 407–412.

[36] A. AlSabeh, E. Kfoury, J. Crichigno, and E. Bou-Harb, "P4DDPI: Securing P4-Programmable Data Plane Networks via DNS Deep Packet Inspection," in *Proceedings of the Network and Distributed System Security Symposium*, 2022.

[37] S. Gupta, D. Gosain, M. Kwon, and H. B. Acharya, "Deep4r: Deep packet inspection in p4 using packet recirculation," in *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*, 2023, pp. 1–10.

[38] L. Teng, C.-H. Hung, and C. H.-P. Wen, "P4SF: A High-Performance Stateful Firewall on Commodity P4-Programmable Switch," in *Proceedings of IEEE/IFIP Network Operations and Management Symposium*, 2022, pp. 1–5.

[39] R. D. Pacífico, L. F. Duarte, L. F. Vieira, B. Raghavan, J. A. Nacif, and M. A. Vieira, "eBPFlow: A Hardware/Software Platform to Seamlessly Offload Network Functions Leveraging eBPF," *IEEE/ACM Transactions on Networking*, 2023.

[40] "AWS Nitro," https://aws.amazon.com/ec2/nitro/, 2024.

[41] "Considerations for large clusters," https://kubernetes.io/docs/setup/best-practices/cluster-large/, 2023.