RDNet: An RDMA-aware Container Network Interface for Cloud Environments

Myoungsung You*, Minjae Seo†, Seungwon Shin‡, and Jaehyun Nam§
*University of Seoul, †ETRI, ‡KAIST, §Dankook University

IEEE INFOCOM 2026 - IEEE Conference on Computer Communications | 979-8-3315-4961-9/26/$31.00 ©2026 IEEE | DOI: 10.1109/INFOCOM59046.2026.11571228

***Abstract*—Remote Direct Memory Access (RDMA) is increasingly adopted in container-based cloud environments for its ability to deliver high-performance communication, making it essential for distributed microservices and AI workloads. However, current container network interfaces (CNIs) face two critical limitations when supporting RDMA: (i) insufficient observability of RDMA operations and (ii) lack of transparent support for legacy TCP/IP containers. RDMA's kernel bypass nature prevents CNIs from monitoring container-level RDMA activities, hindering efficient resource monitoring and exposing multi-tenant environments to security risks. In addition, enabling RDMA for TCP/IP containers requires substantial code modifications due to the differences between RDMA verbs and socket APIs. This paper presents `RDNet`, a novel RDMA-aware CNI that integrates seamlessly with existing container platforms. `RDNet` leverages a user-space eBPF runtime to provide fine-grained RDMA observability at the container level and in-line RDMA policy enforcement with minimal overhead. For TCP/IP containers, `RDNet` relays traffic through a *transparent RDMA proxy* with eBPF-based traffic redirection, avoiding code changes while improving communication performance. Experimental results show `RDNet` incurs less than 5% monitoring overhead, blocks unauthorized RDMA operations, and enhances TCP/IP container throughput by up to 2× for HTTP and gRPC workloads.**

I. Introduction

Containers [1] have become the de facto standard for deploying distributed microservices in cloud environments, offering lightweight virtualization and high scalability. However, the traditional TCP/IP stack, which underpins most container communication, has become a significant performance bottleneck for meeting the demands of large-scale and high-performance microservices [2]. Remote Direct Memory Access (RDMA) [3] provides an attractive alternative by enabling high-throughput, low-latency communication via kernel bypass and NIC hardware offloading [4]–[6]. While integrating RDMA with containers has the potential to deliver substantial performance improvements for microservices, this integration exposes two key challenges in current systems.

Insufficient observability of RDMA operations. RDMA's kernel bypass design fundamentally prevents container network interfaces (CNIs), which manage inter-container connectivity, from monitoring RDMA activity at container granularity. CNIs depend on kernel-based monitoring mechanisms [7], [8], which are incapable of capturing RDMA operations that are executed entirely in user space. As a result, CNIs lack visibility into both the RDMA resource consumption of individual containers and the specifics of inter-container RDMA communication. This lack of observability poses challenges for resource management and security enforcement in cloud environments, as unauthorized RDMA activity can lead to resource exhaustion and facilitate network attacks [9], [10].

Lack of support for TCP/IP containers. RDMA requires applications to use specialized user-space APIs, known as RDMA verbs [11], to directly access NIC hardware. Most containerized applications, however, continue to rely on traditional TCP/IP socket API [12] due to its mature ecosystem. Existing CNIs do not provide mechanisms for these TCP/IP containers to leverage RDMA capabilities, necessitating substantial and error-prone application refactoring to adopt RDMA verbs. This migration is challenging due to the fundamental mismatch between the socket and RDMA programming models.

Previous studies to enable RDMA in container environments [12]–[18] typically address only one of these two challenges and often introduce additional compatibility concerns. FreeFlow [13], for example, provides RDMA observability by inserting a user-space proxy that intercepts RDMA verbs, but this design requires modifications to verb APIs and device drivers, and containers must run with root privileges [19]. TSoR [12] supports TCP/IP containers by translating their socket syscalls into RDMA verbs at the kernel level, but depends on a customized runtime [20], significantly reducing portability across diverse public cloud environments [21].

Our approach. This paper presents `RDNet`, a novel RDMA-aware container network interface that addresses two fundamental challenges in enabling RDMA for containerized workloads: the lack of container-level observability of RDMA operations and the absence of transparent support for TCP/IP-based containers, while maintaining compatibility with existing platforms. `RDNet` leverages eBPF [5] to perform event-driven tracing of RDMA verbs, capturing fine-grained metrics such as queue pair allocations, memory registrations, and data transfer statistics in real time. These insights enable `RDNet` to enforce runtime security policies, blocking unauthorized RDMA operations before they reach the NIC. To sustain high-frequency tracing with minimal performance impact, `RDNet` employs a customized user-space eBPF runtime that avoids repeated kernel crossings. For TCP/IP containers, `RDNet` deploys a lightweight host-level proxy that transparently relays

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2023-00212738) and the IITP (Institute of Information & Communications Technology Planning & Evaluation) - ITRC (Information Technology Research Center) grant funded by the Korea government (Ministry of Science and ICT) (IITP-2026-RS-2023-00258649).

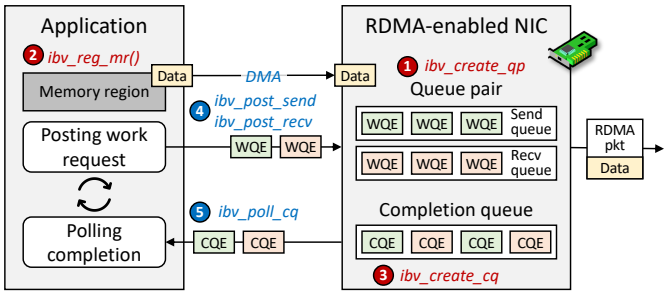

Fig. 1: Common RDMA communication workflow with corresponding control (red) and data (blue) verb APIs.

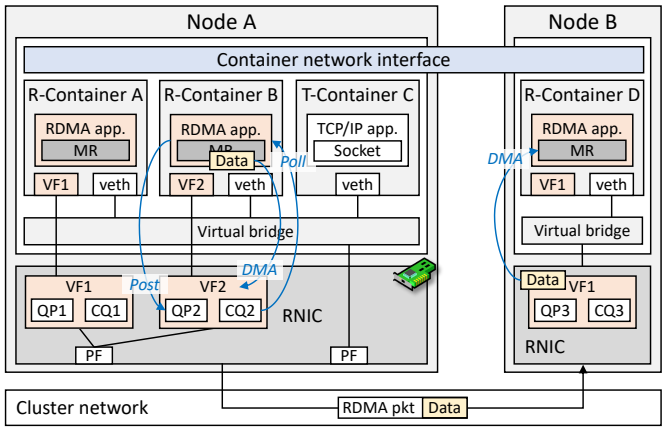

Fig. 2: A common network setup with R- and T-containers.

their traffic over pre-established RDMA connections, using eBPF-based traffic redirection to bypass kernel traversal and improve throughput without code changes. Crucially, RDNet operates alongside standard CNIs and orchestration frameworks, ensuring compatibility with deployment pipelines in large-scale cloud clusters. To the best of our knowledge, this is the first design to combine eBPF-based container-level observability with a fully transparent TCP-over-RDMA relay, providing a practical, deployable foundation for secure, high-performance microservices in modern cloud environments.

Contributions. We make the following contributions:

- We conduct a comprehensive analysis of RDMA networking in containers, highlighting critical limitations in RDMA observability and support for TCP/IP-based containers.
- We design and implement RDNet, an RDMA-aware CNI enabling container-level RDMA observability and transparent TCP/IP acceleration, while maintaining compatibility with underlying container platforms.
- We evaluate RDNet with real-world workloads and show that RDNet incurs less than 5% overhead for RDMA verb tracing, prevents unauthorized RDMA operations, and improves HTTP throughput and gRPC latency by up to 2×.

II. Background and Motivation

A. RDMA Networking

RDMA [3] enables applications across different hosts to exchange data directly without engaging the kernel stack. Complex network tasks (e.g., flow control) are offloaded to the RDMA-enabled NIC (RNIC), and data transfers between an RNIC and applications are performed through zero-copy operations. These optimizations enable RDMA to provide low-latency and high-throughput communication. In RDMA, applications interact with the RNIC through a standardized interface known as RDMA verbs [11]. As shown in Fig. 1, these APIs are classified into two categories: control verb APIs (red ones) and data verb APIs (blue ones).

An application first calls control verb APIs to allocate the necessary resources. As shown in Fig. 1, ① it calls ibv_create_qp to create send and receive queues, known as a queue pair (QP), within the RNIC. The QP represents one endpoint of the RDMA connection, similar to a TCP/IP socket. ② In addition, the application registers a memory region (MR), which maps a section of the host's DRAM to RNIC memory, allowing the RNIC to perform DMA operations on the MR (ibv_reg_mr). ③ Alongside QP and MR, the application creates a completion queue (CQ), which is used to monitor the completion results of data transmissions by polling the CQ (ibv_create_cq). It may use a completion channel to receive notifications through interrupts instead of polling.

Once these RDMA resources are initialized, applications exchange resource identifiers, such as QP numbers (QPNs) and MR addresses, through a third-party channel, typically over TCP/IP. After that, RDMA communication can proceed by using data verb APIs. RDMA supports four data transfer operations: WRITE, READ, SEND, and RECV. The WRITE and READ are one-sided operations, allowing applications to directly WRITE to or READ from a remote MR without involving the remote CPU. The SEND and RECV are two-sided operations that require the sender to post a SEND request (ibv_post_send) and the receiver to pre-post a corresponding RECV request (ibv_post_recv) to its RNIC. Successful data transfer occurs only when both requests are matched. To initiate data transfers, an application posts work queue elements (WQEs) to its QP (④ in Fig. 1). The RNIC processes the WQE, gets data from the registered MR, and transmits an RDMA packet with the data. Upon receiving the packet, the remote RNIC parses it, DMAs the data to the appropriate MR, and sends an ACK packet to the sender's RNIC. Once the ACK packet is received, the sender's RNIC records a new completion queue element (CQE) in the CQ. ⑤ The application retrieves the result by polling the CQ (ibv_poll_cq).

B. RDMA in Container Environments

In container environments, container network interfaces (CNIs) [22] manage connectivity between containers. As shown in Fig. 2, CNIs utilize virtual Ethernet (veth) interfaces to link containers via a virtual bridge on the host side. For enabling RDMA networking, CNIs typically employ Single Root I/O Virtualization (SR-IOV) [23], which creates multiple virtual functions (VFs) from the RNIC's physical function. These VFs operate as independent and virtualized RDMA interfaces, managing their own RDMA resources (e.g., QPs/CQs) while sharing the underlying RNIC hardware.

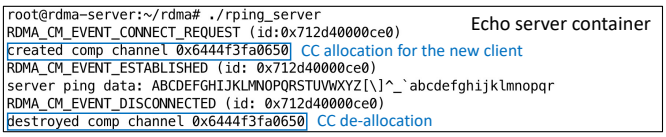

Fig. 3: A response with a benign RDMA communication.

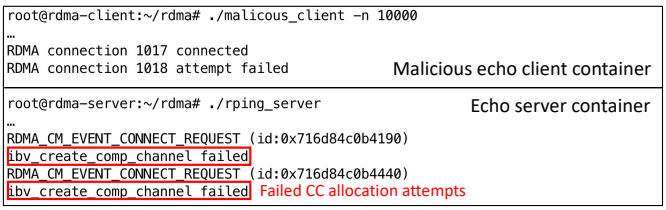

Fig. 4: Server failure due to extensive connection requests.

As shown in Fig. 2, CNIs deploy an extra VF to each container [16], [18], [24]. This setup allows RDMA containers (R-containers), which run applications developed with verbs, to use a dedicated VF for RDMA communication while maintaining the original veth for TCP/IP connections. With the assigned VF, an R-container can issue data verbs and poll completions independently. As shown at the bottom of Fig. 2, R-Container B posts a WQE to QP2, indicating a WRITE operation targeting R-Container D's MR. Before this, R-Containers B and D exchange their RDMA resource identifiers over TCP/IP to establish the RDMA connection. The RNIC on Node A retrieves data from R-Container B's MR using DMA. The RNIC then transmits the RDMA packet containing the data to Node B. Upon receiving the packet, the RNIC on Node B parses it to determine the target VF (VF3). The RNIC DMAs the received data into R-Container D's MR. R-Container B then polls CQ2 to confirm the completion.

C. Limitations of Existing Solutions

We identify that CNIs with SR-IOV face two key limitations that remain unresolved by existing solutions [9], [12]–[16].

Lack of observability on R-containers. To optimize performance, most verb APIs are executed entirely in user space, bypassing the kernel. This kernel bypass design renders traditional kernel-based monitoring mechanisms [7], [8], commonly employed by CNIs, ineffective for verb APIs. Thus, CNIs are unable to observe RDMA operations performed by R-containers. For instance, CNIs cannot determine the amount of RDMA resources (e.g., QPs) allocated to a given R-container, nor can they identify the communication peers of each R-container. This lack of observability introduces security and management risks in multi-tenant environments, where containers belonging to different tenants share RDMA resources. For example, CNIs cannot restrict malicious containers from communicating with restricted endpoints over RDMA [10]. Furthermore, a malfunctioning container may exhaust the host's RDMA resources, hindering other containers from utilizing RDMA [10], [25].

To show this limitation, we conduct an experiment in which a malicious container exhausts RDMA resources. The experimental setup is described in Section IV-B. We deploy an echo server and a client container that communicates using WRITE operations. The server creates a completion channel (CC) for each client to detect new connection requests via interrupts. As shown in Fig. 3, when a benign client connects, the server allocates the new CC and releases it after responding. In contrast, when a client rapidly issues numerous RDMA connection requests without completing the echo protocol, the server reaches a critical threshold: it cannot allocate new CCs after handling 1,017 connections (Fig. 4) and fails to accept new connections. This limitation arises from the verb library's restriction on the number of active CCs. These excessive connection requests are invisible to both the client and server kernels. This example shows that the lack of RDMA observability can lead to security risks in cloud environments.

Lack of support for T-containers. Another major limitation of CNIs is their inability to transparently enable legacy TCP/IP containers (T-containers) to utilize RDMA. While RDMA provides performance benefits, it requires applications to be written using verb APIs. However, most containerized applications continue to rely on TCP/IP socket APIs due to their extensive ecosystem [12]. Enabling T-containers to leverage RDMA would require rewriting these applications to use verb APIs, a task that is effort-intensive due to the fundamental differences between the socket and RDMA programming models. For instance, while socket communication involves simple send and receive operations, RDMA communication requires low-level memory management and polling for completion of one-sided operations. CNIs currently lack mechanisms to bridge this gap, restricting RDMA support to containers explicitly implemented with verb APIs. Consequently, T-containers cannot benefit from the high performance of RDMA, and porting legacy applications remains a labor-intensive process.

Existing solutions. To address these limitations, prior studies [12], [13] have proposed dedicated proxies that intercept verb API calls from containers and provide policy control or RDMA-to-TCP translation. However, none of these approaches overcome both limitations simultaneously. Most require extensive modifications to containerized applications, the verb library, or device drivers to enable proxy interception, leading to significant compatibility issues. Another line of research [14], [15], [26] leverages SmartNICs to achieve RDMA observability at the hardware level. However, these approaches require additional hardware or modifications to RNICs, which raises deployment and compatibility issues. Such limitations have impeded their widespread adoption in cloud environments. More detailed comparisons are provided in Section V. Overall, there remains a pressing need for a transparent solution that simultaneously provides observability for R-containers and RDMA support for T-containers.

III. RDNet Design

A. Threat Model

Similar to prior work on container network security [7], [27], we assume that the cluster infrastructure (e.g., the control plane, nodes, and hardware) is trustworthy. We consider R-containers that use the standard verb library (i.e., libibverbs)

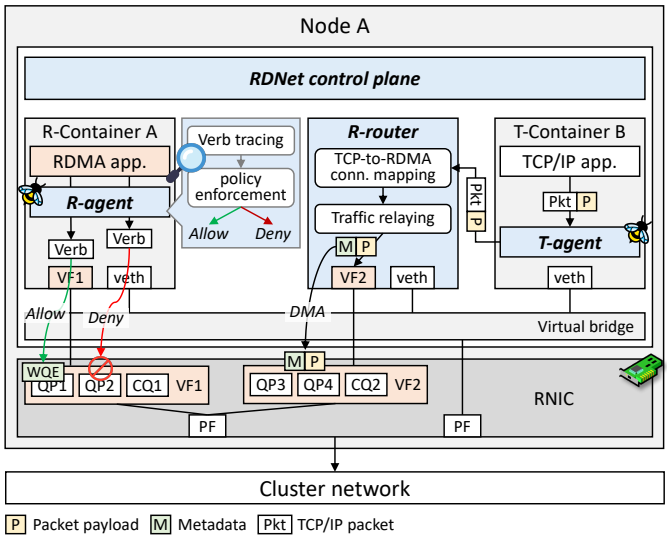

Fig. 5: The overall architecture of RDNet.

to interact with RNICs, and assume that an attacker can compromise such R-containers by exploiting vulnerabilities in RDMA applications running inside them. We exclude container escape attacks [28] and thus consider the host kernel is trustworthy. In this setting, we focus on rule-based policy enforcement for R-containers. Our security goals are to ensure that an attacker cannot (i) perform unauthorized RDMA communication with endpoints outside its permitted scope, or (ii) consume RDMA resources (e.g., QPs) beyond an allowed quota. In this context, we exclude micro-architectural DoS (e.g., RNIC cache thrashing) and side-channel attacks from our scope, as these typically require hardware-level defenses.

B. RDNet Overview

To address the two limitations identified in Section II-C, RDNet employs container-level of eBPF agents and a host-level relay proxy. eBPF [29] enables the safe execution of user-defined programs in kernel space and has been widely adopted in prior work [30]–[32] for system monitoring and efficient packet processing. RDNet leverages eBPF in conjunction with a customized runtime to implement high-performance verb tracing for R-containers and transparent TCP/IP-to-RDMA translation for T-containers.

As shown in Fig. 5, the overall architecture of RDNet comprises three key components: R-agents, T-agents, and the R-router. The **R-agent** is a user-space eBPF program attached to each R-container that implements fine-grained verb tracing. It is triggered upon the invocation of a verb API and inspects both arguments and return values to generate RDMA observability metrics, which are forwarded to the control plane. In addition, the R-agent enforces runtime security policies by rejecting unauthorized verb API invocations in real time, as represented by the Deny path in Fig. 5. Unlike traditional kernel-space eBPF programs, R-agents execute within a custom user-space runtime, avoiding kernel crossing overheads during the inspection of control and data verb APIs and thereby implementing low-overhead tracing. Moreover, as R-agents are managed by the runtime and isolated from containers, verb libraries, and RNIC drivers, RDNet achieves transparent integration with container environments.

TABLE I: The list of verb APIs monitored by the R-agent.

Type	Verb APIs	Observability metrics
Control	ibv_create_qp ibv_destroy_qp	The number of active QPs and QPNs of containers
	ibv_modify_qp	The state of QPs and connections between QPs
	ibv_reg_mr ibv_dereg_mr	The number of active MRs, MR addresses, and MR sizes
	ibv_create_cq ibv_destroy_cq	The number of active CQs and CQNs of containers
Data	ibv_post_send	Tx data sizes, verb types, and remote addresses
	ibv_post_recv	Tx data sizes and local addresses
	ibv_poll_cq	Verb processing results and errors

RDMA support for T-containers is facilitated by the T-agent and the R-router. The **T-agent**, implemented as a kernel-space eBPF program, intercepts socket syscalls issued by T-containers. Upon detecting a connection attempt to a remote T-container, the T-agent redirects the handshake packets to the R-router, thereby establishing a connection with the R-router instead of the intended remote peer. Subsequently, data packets are exchanged between the (local) T-container and the R-router at the socket level, bypassing the kernel network stack. The **R-router**, a per-node relay proxy, maps the TCP/IP connection to a pre-established RDMA connection associated with the destination node. It then relays data packets along with per-packet metadata over RDMA, as shown in Fig. 5. On the receiving node, the R-router forwards the packets to the appropriate T-container based on the metadata. All T-agent operations are confined to kernel space, and communication between T-containers and the R-router remains within the socket layer, avoiding kernel stack overheads. This enables T-containers to transparently leverage RDMA acceleration without modifications to applications and libraries.

C. Fine-grained Observability on RDMA Containers

The R-agent is responsible for enabling fine-grained RDMA observability at the container level by tracing verb APIs through eBPF. While eBPF enables transparent system event tracing, tracing verb APIs with eBPF, particularly data verb APIs, presents a significant performance challenge.

Challenge. R-containers invoke data verb APIs to transmit and receive data over RDMA. Unlike control verb APIs, which involve kernel device driver operations, data verb APIs are executed entirely in user space to optimize performance. However, eBPF programs execute in the kernel space. Thus, tracing data verb APIs with eBPF incurs two kernel crossings per invocation: the *first* occurs when a trap instruction transfers control to the kernel to trigger the eBPF program, and the *second* occurs when control returns to the user-space verb API upon completion of the eBPF program. As demonstrated in our evaluation (§IV-C), this overhead ranges from 2μs to

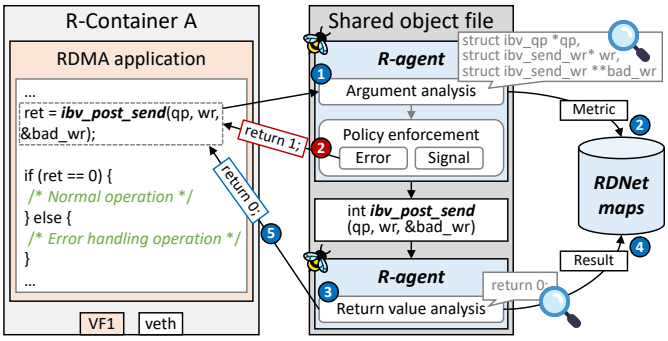

Fig. 6: The workflow of the R-agent.

$4\mu s$. Given that RDMA communication typically completes within a few microseconds, this overhead is substantial.

To overcome this challenge, RDNet employs a customized eBPF runtime that operates in user space rather than kernel space. This runtime enables eBPF programs to trace data verb APIs without incurring kernel crossing overhead, thereby making data verb API tracing by R-agents viable for practical deployment. Beyond performance, the user-space runtime isolates R-agent execution from the kernel, avoids privileged module loading, and supports dynamic hot-swapping per container without application restarts. The RDNet control plane deploys a dedicated R-agent instance (an eBPF program) for each R-container. This R-agent attaches to the shared object file of the verb library within the R-container, as shown by the right part of Fig. 6. Specifically, probe instructions are inserted before and after target verb APIs. The attached R-agent is triggered whenever the R-container invokes control or data verb APIs, enabling it to monitor and control all RDMA operations performed by R-containers.

The scope of our RDMA observability includes both RDMA resource usage and communication details. For resources, RDNet monitors the allocation of QPs, MRs, and CQs to each container. For communication details, it captures the endpoints of connections, operation types (e.g., WRITE), and the volume of data transmitted by containers. Table I summarizes the coverage and the specific verb APIs traced by the R-agent.

RDMA observability. The R-agent's workflow consists of four steps, as shown in Fig. 6. When the application within the R-container calls a verb API, the control flow is forwarded to the R-agent before the called API is executed. ① The R-agent first parses the API's arguments to extract information about RDMA resources or data transfers. For instance, when intercepting the ibv_post_send API, the R-agent parses its argument to identify the transmitted data size, operation type, and the remote R-container. ② This information is then stored in user-space RDNet maps along with the caller container's namespace ID and the process ID. These maps are implemented as shared memory segments in user space, eliminating context switches during map access. After that, the control flow is returned to the called verb API. ③ When the API completes execution, the control flow transitions back to the R-agent, which analyzes the return value to determine the success or failure of the API execution. For example, a non-zero return value for the ibv_post_send indicates a failure, prompting the R-agent to discard the corresponding RDNet map entry, which is inserted in step ②. ④ Otherwise, the R-agent updates the map entry to mark it as valid, ensuring that it can be retrieved by the RDNet control plane. ⑤ Finally, the control flow is returned to the application.

Policy enforcement. The container-level RDMA observability provided by RDNet has various use cases. One of the main use cases is policy enforcement on verb API calls, filtering out potentially malicious calls. For example, RDNet can limit the maximum number of active QPs an R-container can have by enforcing policies on the ibv_create_qp API. Also, it can restrict a container from communicating with only a pre-defined set of containers by examining the ibv_modify_qp() API. For this, the R-agent compares an API call's arguments and the caller container's state against pre-defined policies stored in user-space RDNet maps. As shown by ② in Fig. 6, if the call violates a policy, the R-agent skips the execution of the verb and takes one of two actions: (i) returning an error value (e.g., a non-zero integer) to the caller application, making it appear that the API call has failed; or (ii) sending a termination signal (e.g., SIGKILL) to the caller application, terminating the process. This policy enforcement mechanism enables RDNet to control RDMA operations of R-containers in real time, mitigating security risks.

D. Transparent RDMA Supports for TCP/IP Containers

The R-router is designed to receive data packets from T-containers through TCP/IP connections and relays them over RDMA connections. This design allows T-containers to access RDMA while preserving compatibility with socket APIs, as T-containers and the R-router are connected via TCP/IP connections. However, it incurs substantial per-packet overhead.

Challenge. Containers are deployed within isolated network namespaces. Thus, transmitting a packet from a T-container to the R-router over TCP/IP necessitates traversing the kernel network stack twice: first from the T-container to the virtual bridge, and then from the bridge to the R-router. This process involves (de)packetization and packet copying. Our evaluation shows that these operations incur tens of microseconds of latency per packet, undermining the performance gains of traffic relaying over RDMA. To address this challenge, the T-agent utilizes eBPF-based traffic redirection, which facilitates direct data transfer between T-containers and the R-router, bypassing the kernel stack and the virtual bridge.

Redirecting traffic with eBPF. As shown in Fig. 7, a T-agent is deployed in the kernel space of each T-container and intercepts socket syscalls to perform traffic redirection. ① When a T-container initiates a connection via the connect syscall, the T-agent captures the call and inspects its arguments (i.e., 5-tuple). It checks whether the connection matches policies stored in kernel-space RDNet maps, which administrators configure to specify TCP/IP connections eligible for RDMA relaying. Connections that do not match any policy proceed through the kernel network stack. ② For matching connections, the T-agent rewrites the destination IP address and port

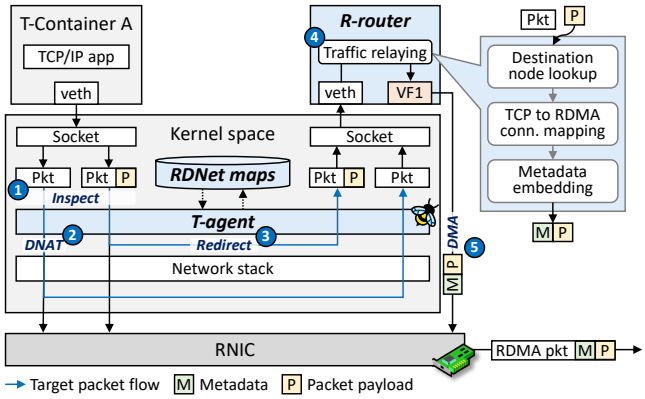

Fig. 7: The workflow of the T-agent and the R-router.

to those of the R-router (i.e., DNAT) and records the original IPs, port numbers, and socket descriptors (SDs) in `RDNet` maps. ③ When the T-container transmits data via a `send` call, the T-agent intercepts the syscall and redirects the data from the T-container's socket to the R-router's socket using the stored SDs. This redirection bypasses the kernel network stack and the virtual bridge, thereby avoiding associated overheads.

Relaying traffic over RDMA. The R-router is responsible for relaying data from T-containers to their destinations via RDMA connections. Upon establishment of a new TCP/IP connection with a T-container, it retrieves the original destination IP and port, which the T-agent has stored in the `RDNet` maps. Since containers communicate using virtual IPs that are not directly routable, the R-router translates the virtual IP of each new connection to the corresponding node's physical IP, as shown in ④ of Fig. 7. The `RDNet` control plane provides up-to-date mapping for this translation. After resolving the target node, the R-router selects an RDMA connection associated with the designated node.

RDMA connections require longer setup times than TCP/IP connections. To avoid connection setup overheads, the R-router employs a connection pooling mechanism. The pool consists of persistent RDMA connections to R-routers on neighboring nodes. When a new TCP/IP connection requires relaying, the R-router selects an available RDMA connection associated with the destination node from the pool. Note that the R-router uses a one-to-one mapping between TCP/IP and RDMA connections. If the number of RDMA-relayed TCP/IP connections reaches the configured maximum, the R-router signals the control plane to update the T-agent policies, preventing further redirections until capacity is available. After selecting a proper RDMA connection, the R-router relays data from the TCP/IP connection along with metadata, including the IDs of the source and destination containers. ⑤ The payload and metadata are then DMAed to the RNIC and transmitted over the selected RDMA connection. On the receiving side, the R-router utilizes the embedded metadata to determine the correct TCP/IP connection with the destination container and relays the payload correctly. Through this design, the R-router relays data between each T-containers over RDMA transparently. The T-agent eliminates per-packet overheads between T-containers and the R-router by intercepting syscalls in kernel space. As a result, T-containers can benefit from RDMA communication without any modifications.

E. Interacting with Container Platforms

The `RDNet` control plane operates as a secondary CNI and manages `RDNet` components and connectivity among R-containers in coordination with the main CNI. When the main CNI configures the network environment for a newly created container, the `RDNet` control plane is automatically invoked by the container platform. It determines whether the container is an R-container or a T-container based on metadata (e.g., labels). For R-containers, the control plane assigns an RNIC-connected VF and configures it using an RDMA subnet manager (e.g., OpenSM [33]), and attaches an R-agent instance to the container's verb library. For T-containers, it deploys a T-agent instance and attaches it to the container's cgroup file system. In addition, the `RDNet` control plane periodically aggregates RDMA metrics from `RDNet` maps populated by R-agents to provide container-level RDMA observability. It also maintains up-to-date cluster state by interacting with the container platform through standard APIs [34], and uses this information to dynamically update policies for both R- and T-agents via user-space and kernel-space `RDNet` maps.

IV. Evaluation

Here, we evaluate the effectiveness and performance of `RDNet` by addressing the following three key questions.

- Q1) Does `RDNet` provide fine-grained observability on R-containers with minimal overhead?
- Q2) What are the main use cases for the container-level RDMA observability provided by `RDNet`?
- Q3) Can `RDNet` transparently accelerate the network performance of T-containers using RDMA?

A. Prototype Implementation

We have developed a full prototype of `RDNet` in C. Specifically, the R-agent is implemented as a bpftime [35] uprobe program and attached via dynamic binary rewriting. Before container launch, the `RDNet` control plane analyzes the target shared object file (i.e., libibverbs.so) within the R-container image to identify the offsets and symbols of the relevant verb APIs. It then inserts inline hooks at these locations. At runtime, when the container invokes a hooked verb API, the inline hook synchronously diverts execution to the user-space eBPF runtime [35] on the same thread, allowing fine-grained monitoring without kernel context switches. The eBPF runtime is implemented on top of ubpf [36] and injected into the running process using ptrace or LD_PRELOAD, eliminating the need for container restarts. The T-agent consists of three eBPF programs (CGROUP_SOCK, SOCK_OPS, and SK_MSG), each executed in the kernel eBPF runtime. These programs are attached to every T-container as well as the R-router. The R-router is implemented using libibverbs [11] and libevent [37], running as a host-level proxy. It runs a main

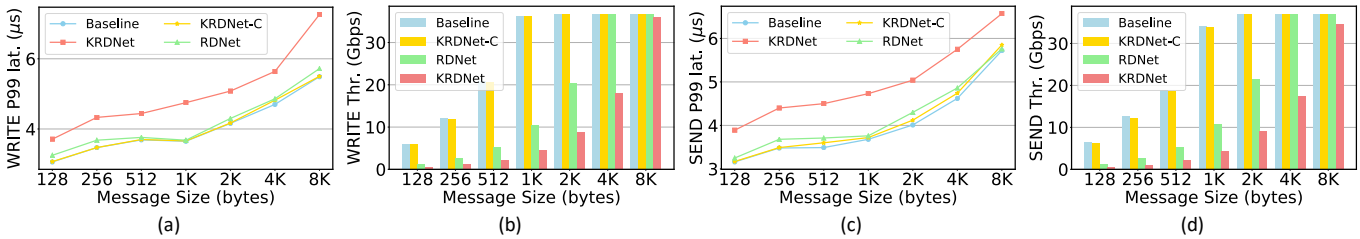

Fig. 8: The performance of RDMA WRITE (a and b) and SEND (c and d) operations on the RC transport mode.

event loop to map connections from T-containers to dedicated threads that manage pre-established RDMA connections. Data exchange between R-routers on different hosts is conducted using WRITE in Reliable Connection (RC) mode [38].

B. Evaluation Environment

We configure a real-world testbed consisting of two physical servers directly connected via a 40 GbE link and configured to operate in RoCE mode. Each server is equipped with a 20-core Intel Xeon 4114 CPU, 64 GB of RAM, and an NVIDIA ConnectX-4 RNIC. Although `RDNet` supports multiple RDMA transport modes (e.g., RoCE and InfiniBand), we focus on RoCE due to its widespread adoption [39], [40]. The container cluster is managed using Kubernetes with the Docker runtime. Flannel [41] is deployed as the primary CNI, while `RDNet` is installed on both servers as a secondary CNI.

C. Overhead of Container-level RDMA observability

To answer the first question, we evaluate the overhead of our verb tracing mechanism by deploying two RDMA `perftest` [42] containers, one on each machine, and measuring the performance of their RDMA communication. The MTU size is set to 1024 bytes, the maximum allowed for RoCE. We compare the performance of four systems: `Baseline`, representing a vanilla RDMA setup; `RDNet`, using the user-space eBPF runtime for control and data verb tracing; `KRDNet`, a variant of `RDNet` that employs the kernel eBPF runtime; and `KRDNet-C`, a variant of `KRDNet` that traces only control verbs, sacrificing observability of data verbs.

Tail latency. Fig. 8(a) and (c) present the tail latency of WRITE and SEND operations between two R-containers, respectively. Measurements of other operations (e.g., READ) are omitted as they show similar results. Across all configurations, the tail latency increases linearly with message size. `KRDNet-C` achieves performance nearly identical to the baseline, as control verbs are involved only during the RDMA connection setup and teardown phases, resulting in negligible overhead during data transmission. In contrast, `KRDNet` incurs a significant latency increase, averaging a 23.3% overhead for all types of operations due to inefficient tracing of data verb APIs. This increase stems from the kernel eBPF runtime introducing two kernel crossings per data verb API invocation, significantly raising tail latency. `RDNet` overcomes this limitation by utilizing the user-space eBPF runtime [35] for tracing data verb APIs, reducing the overhead to less than 5% on average for all data verb types while maintaining real-time observability. These results demonstrate that `RDNet` is suitable for latency-sensitive applications, the primary workload of RDMA, without causing significant performance issues.

Throughput. Fig. 8(b) and (d) present the throughput of WRITE and SEND operations, respectively. Throughput increases linearly with message size across all configurations. For the WRITE operation, the baseline achieves line-rate bandwidth (38Gbps) with messages larger than 1KB, while for the SEND operation, the line rate is reached from 2KB messages onward. `KRDNet-C` performs on par with the baseline, as the overhead of control verb API tracing is negligible. In contrast, `KRDNet` suffers significant throughput degradation due to inefficient data verb API tracing, failing to saturate the line rate even with 8KB messages. `RDNet` shows superior performance, achieving half the line rate with 2KB messages and reaching the line rate with 4KB messages, more than doubling the throughput of `KRDNet`.

The performance gap between the baseline and `RDNet` stems from the limited event-handling capacity of the eBPF runtime in single-threaded applications. In our evaluation, the `perftest` benchmark operates on a single thread, causing the attached eBPF program (R-agent) to execute on the same thread as the data verb API invocations, capping the maximum inspection rate at 1.3M invocations per second and degrading throughput for small messages. However, this limitation arises only when a single thread constantly issues data verb APIs at extremely high rates. When evaluated with a four-threaded version of `perftest`, `RDNet` achieves line-rate throughput for 1KB messages. Since practical cloud deployments involve multi-threaded applications sharing RNIC resources, the likelihood of encountering this bottleneck is minimal.

D. Observability Use Cases

The container-level RDMA observability provided by `RDNet` enables various policy enforcement scenarios. To illustrate `RDNet`'s capabilities, we present two use cases: (1) *Resource Allocation Limit*, which enforces a maximum number of active QPs per R-container; and (2) *Data Transfer Control*, which restricts the types of RDMA operations allowed. Policy enforcement is realized via signal-sending in the first case and error-injection in the second.

Resource allocation limit. `RDNet` provides fine-grained observability into RDMA resource allocation. As shown in Fig. 9(a), it accurately quantifies the RDMA resources allocated to each R-container; for instance, the R-container (ID:4013211150) utilizes two QPs and two CQs. This enables

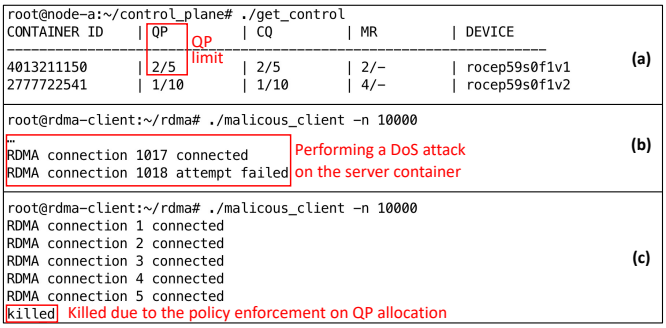

Fig. 9: The RDMA resource observability: RDNet restricts the client container from creating new QPs beyond the limit.

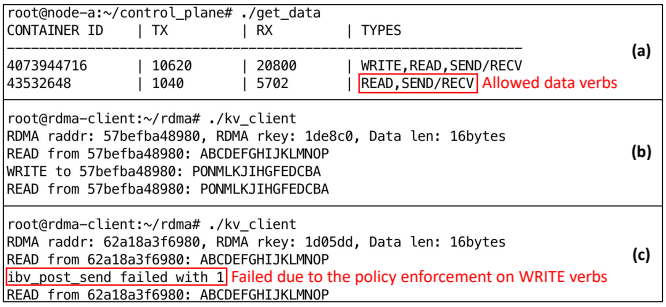

Fig. 10: The RDMA communication observability: RDNet limits the client container from executing WRITE operations.

the enforcement of RDMA resource quotas to prevent resource abuse. Consider the same environment as in our motivating example (Fig. 3 and Fig. 4), where a malicious client attempts to establish an excessive number of RDMA connections with a server. As shown in Fig. 9(b), without RDNet's policy enforcement, the client can allocate unlimited QPs and launch a DoS attack. With policy enforcement enabled, the client is restricted to five QPs, as shown in Fig. 9(c). For this, RDNet intercepts control verb APIs invoked by the client, monitoring the number of active QPs (the red box in Fig. 9(a)) and terminating the client upon exceeding the threshold.

Data transfer control. RDNet provides verb-level observability into inter R-container communication. For instance, as shown in Fig. 10(a), it can aggregate transmitted (TX) and received (RX) bytes at the container level. This enables fine-grained policy enforcement on inter R-container communication. Consider a key-value store scenario (Fig. 10(b)), where a client retrieves data via READ, updates values using WRITE, and then fetches the updated data with another READ. By default, the client performs both READ and WRITE operations without restrictions. When a policy prohibiting WRITE is enforced, the T-agent inspects data verb API calls, identifies the operation type, and blocks WRITE by returning an error code. Consequently, as shown in Fig. 10(c), the client can still perform READ operations but fails to execute any WRITE operations on remote memory.

E. Transparent Acceleration for TCP/IP Containers

Finally, we address the final question by evaluating the performance of T-containers located on different hosts with

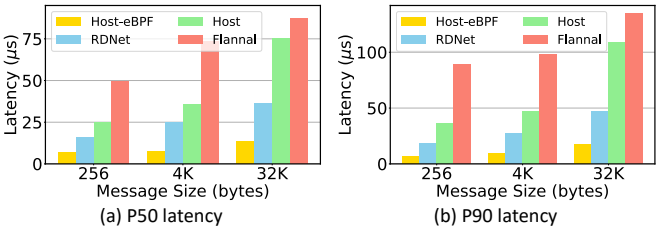

Fig. 11: TCP latency comparison of RDNet.

TABLE II: TCP throughput of MTU-sized packets.

Connections	*Host*	*Flannel*	*RDNet*
1	16.5 Gbps	13.2 Gbps	23.2 Gbps
2	26.2 Gbps	20.6 Gbps	36.5 Gbps
4	36.8 Gbps	23.2 Gbps	36.8 Gbps

and without RDNet. We consider two baselines: the host setting and Flannel CNI. In the host setting, T-containers share the host's network namespace and communicate using the host's IP address instead of virtual IP addresses, avoiding virtualization overheads. In Flannel, T-containers are connected to a virtual bridge and communicate over VXLAN, representing a common container network configuration.

Microbenchmark. We first measure raw TCP/IP throughput and latency using iperf and sockperf, respectively. Table II shows TCP throughput for MTU-sized packets (1450 bytes due to VXLAN) under varying numbers of concurrent TCP connections. With a single connection, RDNet shows a throughput of 23.2 Gbps, outperforming Flannel and the host setting by 1.7× and 1.4×, respectively. While both the host setting and Flannel achieve line-rate bandwidth with four concurrent connections, RDNet saturates the line rate with only two connections. Latency results are shown in Fig. 11. As shown in Fig. 11(a), RDNet achieves a P50 latency of under 40μs for messages up to 32KB, outperforming other baselines regardless of message sizes. A similar trend is observed for tail latency (P90), as shown in Fig. 11(b). For 4KB messages, RDNet exhibits a P90 latency of 36.4μs, outperforming Flannel by 2.9× and the host setting by 1.5×.

These performance improvements are attributed to RDNet's ability to transparently relay TCP/IP packets from T-containers over RDMA connections, offloading data transfer to the RNIC hardware. Furthermore, RDNet eliminates data transfer overheads between T-containers and the R-router by leveraging eBPF-based redirection. The Host-eBPF bar in Fig. 11 shows the latency incurred by this redirection between two containers. Although RDNet experiences slightly higher latency than twice this value due to R-router functionalities, the overall performance remains superior to other baselines.

Application performance. We evaluate two representative applications in cloud environments: a key-value store (Redis) and an RPC streaming server (gRPC), with the server and client containers deployed on separate hosts. Redis workloads are generated using redis-benchmark [43], while gRPC performance is measured with the gRPC benchmark [44].

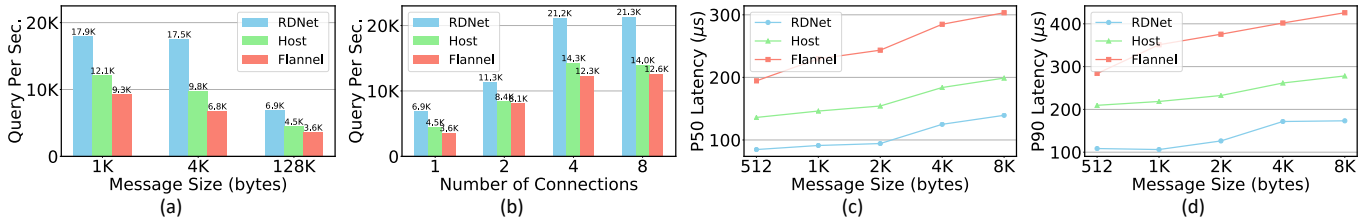

Fig. 12: End-to-end performance of RDNet under Redis (a and b) and gRPC (c and d) communication scenarios.

Throughput is evaluated for Redis and latency for gRPC. Fig. 12(a) presents the throughput of Redis GET operations across various message sizes. Results for SET operations are omitted as they follow similar trends. Across all message sizes, RDNet consistently outperforms both the host and Flannel, achieving up to 2× higher throughput. Fig. 12(b) shows Redis throughput for 128KB messages with varying connection counts. RDNet continues to outperform other baselines by about 1.5×. Fig. 12(c) and (d) show the P50 and P90 latencies for gRPC streaming with 20 concurrent connections. For P50 latency, RDNet achieves latencies below 100μs for messages up to 4KB and 139μs for 8KB messages, representing a 1.4× improvement over the host setting. For P90 latency, RDNet records 173μs for 8KB messages, outperforming Flannel and the host by up to 3.3× and 2×, respectively. These results show that RDNet can transparently accelerate end-to-end network performance for T-containers without requiring any modifications to containerized applications or socket libraries.

V. Related Work

RDMA for containers. Several studies [12], [13], [16], [18], [24], [45] have been conducted to enable efficient RDMA deployment in container environments. FreeFlow [13] employs a centralized proxy that intercepts containers' verb API calls, requiring containers to invoke customized verb APIs that redirect both the call and associated data to the proxy via shared memory channels. The proxy then performs network functions, such as telemetry and access control, before posting the verb operation to the RNIC. TSoR [12] focuses on supporting T-containers by transparently translating socket syscalls into RDMA verbs through a customized container runtime. While these solutions demonstrate the feasibility of RDMA in container environments, they suffer from significant compatibility and security limitations. FreeFlow necessitates modifications to both verb APIs and device drivers, and containers must run with root privileges to communicate with the proxy, introducing security risks. TSoR depends on a specialized container runtime for syscall translation [20], making integration challenging in typical cloud environments where tenants use diverse runtimes. In contrast, RDNet mitigates these limitations by employing an eBPF-based verb tracing that is transparent to containers, verb APIs, and device drivers.

Hardware offloading. Several studies have leveraged SmartNICs [14], [15], [17], [26] or P4 switches [9] to enhance RDMA observability and security. For instance, INSERT [14] and ZETA [15] employ SmartNICs in conjunction with kernel-space eBPF programs to collect host-level RDMA telemetry. Notably, these systems utilize eBPF solely for tracing control verb APIs, since kernel-level tracing of data verb APIs incurs substantial kernel crossing overhead. To overcome this limitation, they rely on SmartNICs to trace individual RoCE packets for data-plane observability. However, these approaches require additional hardware devices, limiting their deployability in cloud environments. Furthermore, they are not designed for containerized environments and lack mechanisms to enable transparent RDMA support for TCP/IP containers. In contrast, RDNet is tailored for containers and adopts a purely software-based approach. It employs the user-space eBPF runtime to provide container-level RDMA observability with minimal overhead and the kernel-space eBPF runtime to enable transparent RDMA support for TCP/IP containers.

VI. Discussion and Limitations

Agent removal. Since the R-agent resides in userspace, an attacker may attempt to restore the original function prologues in *libibverbs.so*. To detect such tampering, the RDNet control plane, running on the host side, monitors the container's memory via *process_vm_readv* [46]. Specifically, it verifies whether the first few bytes of the target verbs still contain the jump instructions directed to the userspace runtime and validates the integrity of the hook handler's code itself to prevent any internal logic subversion. By doing so, RDNet identifies unauthorized removal or modification of hooks and immediately terminates the compromised container.

Customized verb APIs. An attacker may attempt to bypass the R-agent by issuing direct syscalls to the RNIC for control verb APIs. However, since these operations must traverse the kernel, RDNet mitigates this by applying seccomp filters to R-containers. For data verb APIs, which bypass the kernel, an attacker would need to develop a complete, hardware-specific user-space driver and corresponding verb APIs from scratch to bypass the R-agent. Given the extreme complexity and vendor-specific nature of RNIC doorbell mechanisms, such an approach is highly impractical for real-world scenarios.

VII. Conclusion

This paper presents RDNet, a novel RDMA-aware CNI that enables fine-grained RDMA observability and transparent TCP/IP-to-RDMA translation for containers. RDNet leverages user-space eBPF-based verb tracing for R-containers and combines eBPF-based traffic redirection with an RDMA proxy to accelerate T-containers. Experimental results demonstrate that RDNet provides low-overhead RDMA monitoring while effectively improving the performance of T-containers.

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] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 65–77.

[3] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance {RDMA} systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 437–450.

[4] J. Shu, Y. Chen, Q. Wang, B. Zhu, J. Li, and Y. Lu, "Th-dpms: Design and implementation of an rdma-enabled distributed persistent memory storage system," *ACM Transactions on Storage (TOS)*, vol. 16, no. 4, pp. 1–31, 2020.

[5] J. Xue, Y. Miao, C. Chen, M. Wu, L. Zhang, and L. Zhou, "Fast distributed deep learning over rdma," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–14.

[6] F. Tian, Y. Zhang, W. Ye, C. Jin, Z. Wu, and Z.-L. Zhang, "Accelerating distributed deep learning using multi-path rdma in data center networks," in *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2021, pp. 88–100.

[7] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin, "{BASTION}: A security enforcement network stack for container networks," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 81–95.

[8] Isovalent, "High Performance Cloud Native Networking (CNI)," https://www.cilium.io/, 2024.

[9] J. Xing, K.-F. Hsu, Y. Qiu, Z. Yang, H. Liu, and A. Chen, "Bedrock: Programmable network support for secure {RDMA} systems," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2585–2600.

[10] B. Rothenberger, K. Taranov, A. Perrig, and T. Hoefler, "{ReDMArk}: Bypassing {RDMA} security mechanisms," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 4277–4292.

[11] Linux, "RDMA userspace verb library," https://github.com/linux-rdma/rdma-core, 2024.

[12] Y. Sun, Q. Qu, C. Zhao, A. Krishnamurthy, H. Chang, and Y. Xiong, "Tsor: Tcp socket over rdma container network for cloud native computing," *arXiv preprint arXiv:2305.10621*, 2023.

[13] D. Kim, T. Yu, H. H. Liu, Y. Zhu, J. Padhye, S. Raindel, C. Guo, V. Sekar, and S. Seshan, "{FreeFlow}: Software-based virtual {RDMA} networking for containerized clouds," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 113–126.

[14] H. Chang, W. A. Hanafy, S. Mukherjee, and L. Wang, "Insert: In-network stateful end-to-end rdma telemetry," in *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024, pp. 1061–1070.

[15] H. Chang and S. Mukherjee, "Zeta: Transparent zero-trust security add-on for rdma," in *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024, pp. 1041–1050.

[16] Y. Zhang, Y. Tan, B. Stephens, and M. Chowdhury, "Justitia: Software {Multi-Tenancy} in hardware {Kernel-Bypass} networks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 1307–1326.

[17] Q. Su, C. Wang, Z. Niu, R. Shu, P. Cheng, Y. Xiong, D. Han, C. J. Xue, and H. Xu, "Pipedevice: a hardware-software co-design approach to intra-host container communication," in *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*, 2022, pp. 126–139.

[18] W. Liu, K. Qian, Z. Li, F. Qian, T. Xu, Y. Liu, Y. Guan, S. Zhu, H. Xu, L. Xi *et al.*, "Mitigating scalability walls of {RDMA-based} container networks," in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, 2025, pp. 1049–1065.

[19] Microsoft, "FreeFlow Github Repository," https://github.com/microsoft/Freeflow, 2024.

[20] C. Zhao, Y. Sun, Y. Xiong, and A. Krishnamurthy, "Quark: A high-performance secure container runtime for serverless computing," *arXiv preprint arXiv:2309.12624*, 2023.

[21] R. Kumar and B. Thangaraju, "Performance analysis between runc and kata container runtime," in *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*. IEEE, 2020, pp. 1–4.

[22] C. N. C. Foundation, "CNI - the Container Network Interface," https://github.com/containernetworking/cni, 2024.

[23] NVIDIA, "Getting Started with OpenShift," https://docs.nvidia.com/networking/display/kubernetes2440/getting-started-openshift.html, 2024.

[24] Z. He, D. Wang, B. Fu, K. Tan, B. Hua, Z.-L. Zhang, and K. Zheng, "Masq: Rdma for virtual private cloud," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 1–14.

[25] X. Kong, J. Chen, W. Bai, Y. Xu, M. Elhaddad, S. Raindel, J. Padhye, A. R. Lebeck, and D. Zhuo, "Understanding {RDMA} microarchitecture resources for performance isolation," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 31–48.

[26] K. Taranov, B. Rothenberger, A. Perrig, and T. Hoefler, "{sRDMA}–efficient {NIC-based} authentication and encryption for remote direct memory access," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 691–704.

[27] M. You, J. Nam, M. Seo, and S. Shin, "Helios: Hardware-assisted high-performance security extension for cloud networking," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, 2023, pp. 486–501.

[28] Z. Jian and L. Chen, "A defense method against docker escape attack," in *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, 2017, pp. 142–146.

[29] 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 Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[30] W. Findlay, D. Barrera, and A. Somayaji, "Bpfcontain: Fixing the soft underbelly of container security," *arXiv preprint arXiv:2102.06972*, 2021.

[31] J. Jia, Y. Zhu, D. Williams, A. Arcangeli, C. Canella, H. Franke, T. Feldman-Fitzthum, D. Skarlatos, D. Gruss, and T. Xu, "Programmable system call security with ebpf," *arXiv preprint arXiv:2302.10366*, 2023.

[32] W. Findlay, A. Somayaji, and D. Barrera, "Bpfbox: Simple precise process confinement with ebpf," in *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2020, pp. 91–103.

[33] NVIDIA, "OpenSM - Subnet Manager," https://docs.nvidia.com/networking/display/winofv55054000/opensm+-+subnet+manager, 2024.

[34] Kubernetes (K8s), "The Kubernetes API," https://kubernetes.io/docs/concepts/overview/kubernetes-api/, 2024.

[35] Y. Zheng, T. Yu, Y. Yang, Y. Hu, X. Lai, and A. Quinn, "bpftime: userspace ebpf runtime for uprobe, syscall and kernel-user interactions," *arXiv preprint arXiv:2311.07923*, 2023.

[36] IOVisor, "uBPF, Userspace eBPF VM," https://github.com/iovisor/ubpf, 2025.

[37] Libevent, "Libevent: An Event Notification Library," https://libevent.org/, 2012.

[38] NVIDIA, "RDMA Aware Networks Programming User Manual: transparent modes," https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17/transport+modes, 2024.

[39] W. Li, J. Zhang, Y. Liu, G. Zeng, Z. Wang, C. Zeng, P. Zhou, Q. Wang, and K. Chen, "Cepheus: accelerating datacenter applications with high-performance roce-capable multicast," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 908–921.

[40] T. Khan, S. Rashidi, S. Sridharan, P. Shurpali, A. Akella, and T. Krishna, "Impact of roce congestion control policies on distributed training of dnns," in *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 2022, pp. 39–48.

[41] Flannel, "Flannel CNI," https://github.com/flannel-io/flannel, 2024.

[42] perftest, "Open Fabrics Enterprise Distribution (OFED) Performance Tests," https://github.com/linux-rdma/perftest, 2024.

[43] Redis, "Redis benchmark," https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/benchmarks/, 2024.

[44] gRPC, "Overview of performance test suite," https://github.com/grpc/grpc/tree/master/tools/run_tests/performance, 2024.

[45] U. Abbasi, E. H. Bourhim, M. Dieye, and H. Elbiaze, "A performance comparison of container networking alternatives," *IEEE Network*, vol. 33, no. 4, pp. 178–185, 2019.

[46] Linux, "process_vm_readv(2) - Linux man page," https://man7.org/linux/man-pages/man2/process_vm_readv.2.html, 2026.