# A Flexible and Efficient Container-based NFV Platform for Middlebox Networking

Chao Zheng
School of Cyber Security, UCAS
Institute of Information Engineering, Chinese Academy of Sciences

Qiuwen Lu*
Institute of Information Engineering, Chinese Academy of Sciences

Jia Li
Institute of Information Engineering, Chinese Academy of Sciences

Qinyun Liu
Institute of Information Engineering, Chinese Academy of Sciences

Binxing Fang
Institute of Electronic and Information Engineering of UESTC in Guangdong

## ABSTRACT

Network Function Virtualization (NFV) enables multiple network functions (NFs) to operate simultaneously on a commodity server. Internet Data Centers (IDCs) gain significant flexibility and agility through NFV's ability to dynamically deploy and terminate virtual NFs. However, NFV has complicated the middlebox deployment dependence on topology. In addition, NFV requires dramatically higher network throughput on commodity hardware. Current approaches to high performance I/O platforms such as Intel's Data Plane Development Kit (DPDK) and Netmap enable high network throughput, but these were designed for a single dedicated NF. To address these issues, we propose a high-performance platform based on Docker containers and DPDK for the deployment of multiple virtual middleboxes. The platform provides NFs with a higher abstraction layer for the underlying hardware, to facilitate NF deployment, packet processing, and inter-NF communication. Our evaluation shows that the platform provides proper isolation of NFs with 4% overhead. For a service chain with numbered NFs, our solution outperforms the Single Root I/O Virtualization (SR-IOV) platform with 7× the throughput.

## CCS CONCEPTS

• **Networks** → **Middle boxes / network appliances**;

## KEYWORDS

NFV, docker, DPDK, service chain, middlebox

*Corresponding author: luqiuwen@iie.ac.cn.


*SAC 2018, April 9–13, 2018, Pau, France*

ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167240

**ACM Reference format:**
Chao Zheng, Qiuwen Lu, Jia Li, Qinyun Liu, and Binxing Fang. 2018. A Flexible and Efficient Container-based NFV Platform for Middlebox Networking. In *Proceedings of SAC 2018: Symposium on Applied Computing , Pau, France, April 9–13, 2018 (SAC 2018),* 7 pages.
https://doi.org/10.1145/3167132.3167240

## 1 INTRODUCTION

The proliferation of Network Function Virtualization (NFV) has attracted significant attention by bringing greater openness and agility to network data planes. The NFV network model focuses on the implementation of software-based network functions (NFs) that are run on a virtualized infrastructure. This concept enables multiple virtualized NFs to operate simultaneously on the same high-volume server. As a consequence, Internet Data Centers (IDCs) have gained significant flexibility and agility because of their ability to dynamically deploy and terminate virtual NFs. When multiple NFs run simultaneously and each of these processes a large volume of network traffic, network I/O performance is a concern. To address this concern, hardware and software techniques are adopted to enhance NF performance, including Single Root I/O Virtualization (SR-IOV), Open vSwitch [11], Intel Data Plane Development Kit (DPDK)[4], and Virtio [13]. These approaches mainly focus on optimizing the end host's network performance and, in particular, switching packets to different virtual machines (VMs) efficiently. However, middleboxes like firewalls, Intrusion Detection Systems (IDSs), Deep Packet Inspection filters (DPIs), and Data Leakage Prevention systems (DLPs), which often share the same packets or forward packets to one another, can also run on the same hypervisor, and their special needs are rarely considered.

In this paper, we describe and evaluate the Multiple Virtual Middlebox Platform (MVMP), an NFV platform that is based on Docker containers and Intel's DPDK, extended to support flexible deployment, fault isolation, and interoperability of multiple NFs. MVMP provides a higher-level abstraction to facilitate packet processing by multiple middleboxes on one hypervisor, particularly addressing the needs

of DPIs and IDSs. The platform runs as a daemon in Linux, and each NF is packaged as a Docker container for easy deployment and resource isolation. MVMP takes advantage of DPDK's high-throughput packet processing capabilities, so that NFs on the MVMP platform can work at line speed (10Gbps) using commodity hardware.

In this work, we focus on three network virtualization capabilities:

The **flexibility** to create and deploy NFs in different environments. Modern IDCs apply techniques such as the Virtual Extensible LAN (VXLAN) or the Network Virtualization using Generic Routing Encapsulation (NVGRE) to create a huge virtualized Layer 2 network, basically by wrapping an IP header around the tenant's packet. However, in practice, this complicates the deployment of passive network traffic analyzers such as IDSs and DPIs, as they must be deployed at specific positions to access unwrapped traffic. Our major concern is to allow NF deployment to be independent of the details of the underlying network.

The **efficiency** of inter-NF communication. Middleboxes like IDSs and DLPs are often deployed together and share traffic to achieve the best protection. In addition, NFs must be able to migrate packets to each other. For example, a router may forward specific packets to an Intrusion Prevention system (IPS) when it is under a Distributed Denial of Service (DDOS) attack, and the IPS must decide whether the packet needs to be allowed through or throttled. We seek to meet the efficiency requirement in MVMP by providing an efficient forwarding mechanism that does not copy packets.

The **robustness** of the platform. It should be able to isolate NF failures, e.g., random crashing, and prevent mutual interference. An NF like a DPI with an experimental module may be very unstable, and some unforeseen flows might crash the NF. Worse yet, a packet buffer that is shared among NFs might be overwritten.

To provide the desired capabilities, MVMP includes the following innovations:

- A container-based platform for flexible middlebox deployment that combines Docker containers and DPDK. Moreover, it can match the performance requirements on customized hardware.
- A network interface abstraction layer that shields NFs from the details of the underlying network and hardware. For example, if a DPI wants to access unwrapped packets in a VXLAN environment, a VXLAN gateway can be deployed upstream to decapsulate the packets. In addition, the virtual device concept in the abstraction layer simplifies the organization of NFs in a service chain.
- A shared-memory mechanism that allows NFs running on the same hypervisor to share and forward network traffic efficiently and still retain the necessary protection.

We compare MVMP with the NFV platforms SR-IOV and OpenNetVM [16], a state-of-the-art NFV platform. In a service chain experiment using 5 NFs, MVMP throughput outperformed SR-IOV and OpenNetVM by as much as 7× and 3×, respectively.

## 2 CONSTRAINTS AND RATIONALE

The operating environment for a virtual NF is different from that of a traditional network appliance. In what follows, we briefly discuss constraints and challenges stemming from these differences, both to reveal the rationale behind the design choices of MVMP and to highlight what makes it unique.

### 2.1 Traditional Middleboxes vs. NFV Middleboxes

In traditional networks, a service chain includes a set of network appliances offering services such as load balancers, firewalls, DPIs, IDSs, and more, to support dedicated networking processing and applications. An NFV-based middlebox must offer functionality and semantics equivalent to those of an onsite middlebox—i.e., firewalls must drop packets correctly, IDSs must trigger identical alarms, etc. In contrast to traditional endpoint applications, this is challenging because middlebox functionality is usually dependent on the network topology.

As Han [5] summarized, NFV poses several challenges and opportunities for network operators, such as a network performance guarantee for virtual appliances, their dynamic instantiation and migration, and their efficient placement. In addition, NFV complicates the middlebox deployment dependence on topology. For geographically distributed networks, NFV uses a series of new technologies to decouple NFs from their location. These technologies all introduce new tags or new wraps in packets and new translation endpoints. For example, VXLAN [8] adopts VXLAN tags and VXLAN Tunnel End Points (VTEPs), and NVGRE [14] uses a similar idea. Middleboxes' deployment locations are constrained , as they must access unwrapped network traffic. Furthermore, as multiple middleboxes in a service chain share the same hardware and network flow, researchers have an opportunity to optimize the traffic scheduling. Packet forwarding on the same hypervisor could be performed in memory instead of sending the packets through network interface cards (NICs) and switches, and so could traffic sharing among NFs.

OpenNetVM provides a mechanism for steering network traffic between NFs. It uses a flow table that directs packets between the NFs in service chains; this table can be configured dynamically by NFs. However, the solution of steering traffic with a flow table is not adequate. First, placement of modern NFs on the network topology is complicated, as they can be inline—e.g., routers—or bypass—e.g., DPIs. For a service chain containing the two types of NFs, routing packets based on a flow table can be intricate. Second, packets in a service chain are not only forwarded through NFs, but also modified. To process packets from VXLAN, for example, an

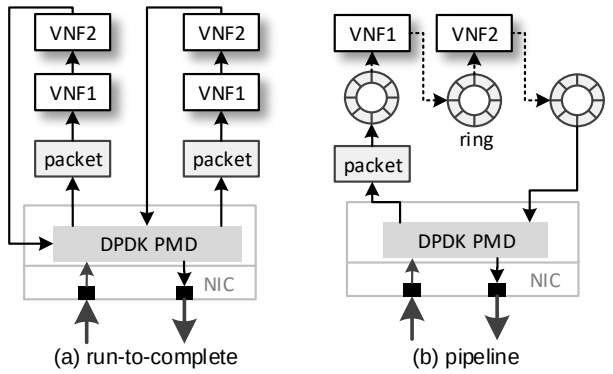



**Figure 1: Two programming models for packet processing.**

NF that is upstream of a service chain may need to decapsulate the packet before forwarding it. MVMP solves these problems by providing an abstract network interface layer.

## 2.2 Fast-packet I/O vs. Isolation

Network performance for NFs on Commercial Off-The-Shelf (COTS) hardware has been a concern since long before NFV emerged. Several technologies have been developed to achieve fast I/O on COTS hardware, such as Intel's DPDK, `Netmap` [12], `PF_RING` [3] and GPU-accelerated approaches [6, 15]. Among these technologies, the authors chose DPDK as the I/O infrastructure for MVMP. As a fast user-space packet I/O engine, its performance and stability have been proven by researchers and companies. Compared to `Netmap` and `PF_RING`, DPDK supports more hardware and has a thriving community of users.

DPDK is a set of data plane libraries and network interface controller drivers for fast packet processing with an inexpensive, commodity X86 server [1]. DPDK implements a low overhead Run-to-Completion (RTC) model for fast data plane performance and accesses devices via polling to eliminate the performance overhead of interrupt processing. DPDK provides many data structures for developing high performance NFs. There are two programming models for middleboxes, namely, RTC and pipelining. Figure 1 intuitively presents the two models. RTC is the classical fast-packet processing model, and most of the DPDK demos and benchmarks are written in accordance with the RTC model, e.g., l2fwd and l3fwd. In RTC, each cycle consists of packets being received by the Poll Mode Driver (PMD) and a series of callbacks. For example, a firewall programmed with the RTC model starts its packet processing cycle by acquiring a packet via the PMD, then examines the packet header and content, and ends with an action of either forwarding or discarding the packet, based on the result of the examination. The pipeline programming model uses queues to transfer packets through different cores. For example, a firewall programmed with a pipeline has a packet-receiving thread, an examination thread, and a forwarding thread, which communicate with each other through lockless queues. This gives users the flexibility to develop asynchronous applications.

Along with other fast packet I/O technologies, DPDK was designed mainly for a single NF. Thus, developers need to do extra work to deploy multiple tenants that share the same NIC on one hypervisor. Basically, there are three possible ways to develop an NFV platform that supports multiple tenants. First, NFs can be carefully crafted to callback style and integrated with an RTC process, which is inconvenient and fragile. Second, SR-IOV can be used to allow multiple processes to acquire packets from one physical interface. The shortcoming of this method is that packet switching can only be performed based on the Layer 2 address. Third, a shared-memory-based pipeline can be used, in which one producer process moves packets from the NIC to the pipeline and several consumer processes properly handle these packets. This method is flexible and efficient enough to build a multi-tenant platform and can also provide proper isolation between NFs.

However, in a production environment, multi-process applications like those built using the third of these methods have to deal carefully with inter-process synchronization. This is because processes communicate by a huge shared page memory, which is delicate and fragile. If a consumer process randomly crashes, the status of shared memory may be corrupted. This should not be a problem if there is only one consumer, for we can simply reboot the producer to retrieve a correct status. But when there are a number of consumers, rebooting is infeasible. Therefore, MVMP adopts a shared-memory framework that retains the necessary isolation between NFs.

# 3 SYSTEM DESIGN

As shown in Figure 2, the MVMP architecture consists of a virtual device abstraction, a shared memory mechanism, and a control plane. The platform runs as a Linux daemon; it has several work threads that poll packets from NICs with DPDK's PMD and then dispatches them to virtual devices. Packet buffers reside in the shared memory, which allows access by multiple NFs in different processes. Each NF runs as a user-space process inside a Docker container instead of a VM, which makes them lighter weight and still isolated. The NFs send and receive packets by invoking mvAPI (short for MVMP API), which allows an NF to receive multiple packets during one invocation, so NFs can use batching to amortize costs. The control plane is used for NF status and ARP table maintenance.

Figure 2 depicts a service chain built by three NFs. A firewall (NF1) is deployed as an inline NF that handles packets through a virtual device (VD1). The packet buffer resides in the shared memory. A DPI (NF2) intercepts at VD1 and works as a bypass NF, sending suspicious traffic to an IPS (NF3) through a virtual device (VD2) that does not map to any physical NIC. The IPS (NF3) builds a TCP reset

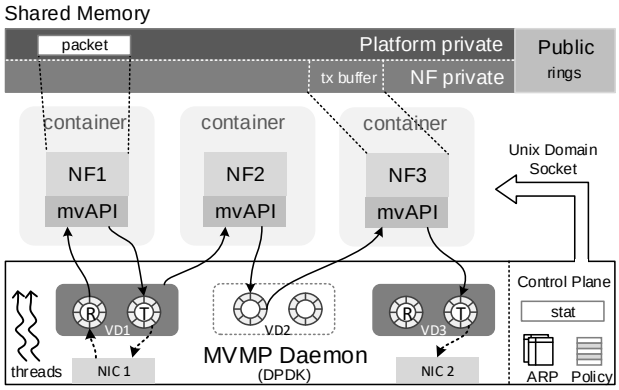



**Figure 2: An overview of MVMP.**

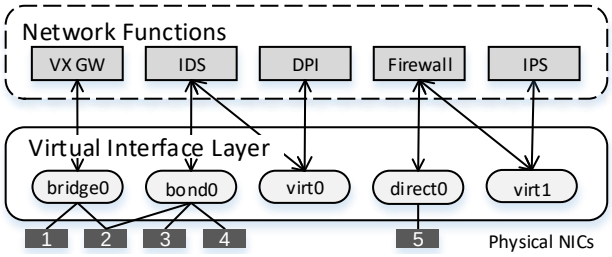



**Figure 3: Example of NIC organization: A VXLAN Gateway (VX GW) translates traffic through two bridged NICs, an IDS inspects three NICs' inbound and outbound traffic and sends some suspicious traffic to a DPI through virt0, and a firewall filters the packets of one NIC and also cooperates with an IPS through virt1.**

packet with a buffer in shared memory and injects the packet through VD3 to stop malicious connections.

### 3.1 Virtual Devices

It is common for a hypervisor to have several NICs to increase port density. The NICs are grouped with different logics to accomplish different middlebox functions. Usually, there are three primary ways to organize NICs. First, a single NIC can be used as a single device, e.g., one-armed routers and IDSs. Second, NICs can be bonded to increase the available bandwidth, e.g., Content Delivery Network (CDN). Third, NICs can be bridged for forwarding, e.g., firewalls and routers. In addition, the network traffic of one NIC may be load balanced to or shared by multiple NFs. For example, an IDS will need to share a firewall's traffic to detect potential threats.

Focusing on these requirements, we propose a virtualization layer on top of DPDK. The virtualization layer abstracts from the physical NICs and anchors the NFs to the virtualized device. This ensures that NF lifecycles are independent of the allocation and organization of underlying NICs. The NICs, virtual devices, and NFs are organized in accordance with the following principles:

1. A virtual device is an abstraction of $N$ NICs, $N \geq 0$.
2. If $N > 0$, the NICs are organized as bonded, bridged, or directly connected.
3. If $N = 0$ , the virtual device works like a bidirectional queue, namely, a rootless device. The rootless device enables inter-NF communication.
4. One physical NIC can belong to multiple virtual devices.
5. One virtual device can be opened by multiple NFs and receive duplicate inbound traffic.
6. One NF can open multiple virtual devices.

Figure 3 illustrates a use of MVMP that includes five NFs and two service chains.

Packets initially arrive in a queue on an NIC port, and then the worker threads invoke DPDK's PMD, which ensures that packet data are stored directly into the shared memory pool through Direct Memory Access (DMA). The worker threads are a part of the MVMP daemon; their number depends on the workload. These threads examine packets individually to decide which virtual devices they should be dispatched to, based on a set of predefined policies. Note that only the packet descriptor is copied and placed in the virtual device's receive queue. The dispatching policies describe which packets a virtual device will receive and how they are encapsulated and decapsulated. In more detail, the policies are as follows:

- Receiving policies describe desired packets, including the packet size, protocol, address, and port and combinations thereof.
- Packets' encapsulation and decapsulation instructions are used for stateless protocols, e.g., jump over the VLAN and GRE tags and deliver to the DPI with an inner IPv4 header. Because passive analyzers such as DPIs and IDSs usually don't send packets, they may be more portable because they do not need to deal with the encapsulating tags.
- Sending policies describe which NIC to use. For example, a bonded device may choose NICs in a round-robin fashion, while a bridged device may direct packets to the opposite NIC from which it was received.

**Table 1: Policy maps for NICs and virtual devices**

| | bridge0 | bond0 | direct0 |
|---|---|---|---|
| NIC1 | VXLAN | – | – |
| NIC2 | 192.168.0.0/16 | 192.168.0.0/8 | – |
| NIC3 | – | all | – |
| NIC4 | – | all | – |
| NIC5 | – | – | vlan1/2 |

Table 1 shows the dispatching policy for Figure 3: the virtual devices declare their receive policies in columns, and the MVMP daemon executes the policies by row.

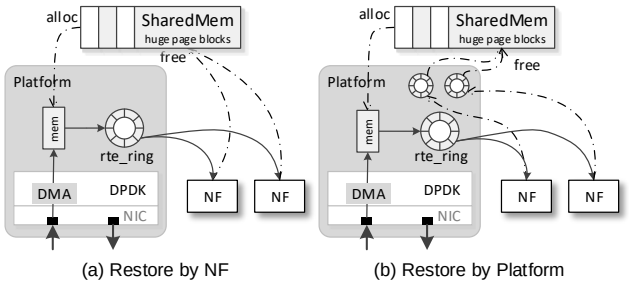



**Figure 4: Two methods for restoring a packet buffer.**

From the perspective of the platform, each NF has an independent instance of a virtual device, and the platform adds the packet descriptor to the instance's receive queue, thereby allowing an authorized NF to have a full view of the virtual device's traffic. A service chain is self-organizing by NFs forwarding and receiving packets through a rootless virtual device. NFs steer traffic by choosing a virtual device to forward each packet to. An upstream NF such as a router mitigates DDoS flooding traffic to IPSs. Forwarding is accomplished by NFs instead of the platform.

### 3.2 Shared memory framework

A typical DPDK pipeline application is constructed by a huge shared page memory pool and a circular queue. The platform cores move packets from NICs to shared memory via DMA and then deliver the pointer to the packet buffer to the NF cores by `rte_ring`. After handling the packet appropriately, the NFs will restore the buffer to the original memory pool. This procedure, which is shown in Figure 4(a), works fine for a single process.

The problem comes when there are multiple separate NF processes. First, if one NF crashes while restoring the packet buffer, this may corrupt the memory pool and jam the platform and other NFs. Second, both the platform and the NFs operating on the memory pool will cause more write contentions, which will decrease performance. Third, a shared packet might be overwritten by a careless NF, which could be a security concern.

To address these problems, we assigned a dedicated ring to each NF to restore the packet buffer. As shown in Figure 4(b), each NF places the buffer descriptor in its dedicated ring, and then the platform restores the buffer to the memory pool. For a single consumer and single producer scenario, `rte_ring` is lockless. Moreover, since the NFs no longer directly operate on the memory pool, it is possible to set packet buffers as read-only for non-trusted NFs, thereby avoiding unintentional overwrites. In this way, all three of the aforementioned problems are solved. As shown in Figure 2, the global shared memory can be partitioned into three categories based on the NFs' access privileges (note that the platform has full control of the entire memory):

1. **Platform private memory** is used for platform DMA packet data from NICs and is read-only to NFs.
2. Each NF has an **NF private memory**, which is used for constructing packets without a memory copy. The owner is permitted to read and write to other NFs depending on a memory view authorization table, where authorized NFs have the privilege to read/write/allocate/deallocate private memory space. The platform also uses this memory to clone indirect packets, so that NFs can operate on the offset of the packet buffer and skip protocol headers.
3. **Public memory** is used to transfer packet descriptors, through queues and rings. NFs can access this memory with mvAPI.

### 3.3 Control plane

MVMP uses UNIX domain socket for communication with NFs, such as NF statuses and control messages. After initializing the platform daemon, MVMP starts a thread that listens on a UNIX domain socket for new NF connections. NFs use this connection to notify the platform which virtual device to open, with configurations like that in Table 1 and heartbeat messages. When a guest NF is in a deadlock or has crashed, MVMP should inform each virtual device and recycle resources. A closing or SIGPIPE event for that connection indicates an NF failure, so that the platform can inform each virtual device and do some cleanup. If an NF works on Layer 3, e.g., as a proxy, it needs to perform an ARP resolution. Since virtual devices are shared by multiple NFs, the ARP operation is processed by the platform, and the ARP cache is synchronized with the connections between NFs and the platform. When an NF fails to look up an entry from the ARP cache, a control message is sent to the platform. After the platform completes the ARP request, the NF's ARP cache will be updated by the next message. In conclusion, MVMP uses control messages to synchronize NFs' dynamic structures instead of using shared memory, and this method is briefer and simpler.

## 4 EVALUATION

In this section, we evaluate MVMP's performance with respect to traffic duplication and forwarding, which are used for bypassed and inlined NFs, respectively. We evaluate service chain performance of MVMP by comparing it with SR-IOV and OpenNetVM. All three approches were deployed on a commodity server with an Intel Xeon CPU E5-2650 v2 @ 2.60GHz (16 physical cores, 16×2 logical cores), a 64GB DDR3 1333Mhz memory, and an Intel 82599ES 10G NIC. The server runs Linux CentOS 7.2 (kernel 3.10.0). We used an IXIA traffic generator [2] for packet generation.

### 4.1 Overhead

**Packet Forwarding:** As MVMP uses extra rings to deliver packets, we evaluated the overhead by running a packet-forwarding NF. Both MVMP and OpenNetVM use two logical CPU cores, one for the daemon and another for the NF. DPDK l2fwd uses two logical CPU cores. Figure 5 shows that l2fwd reaches 14.3 million packets per second (Mpps) when forwarding 64-byte packets, MVMP sees no drop, and the number of packets processed by OpenNetVM falls by

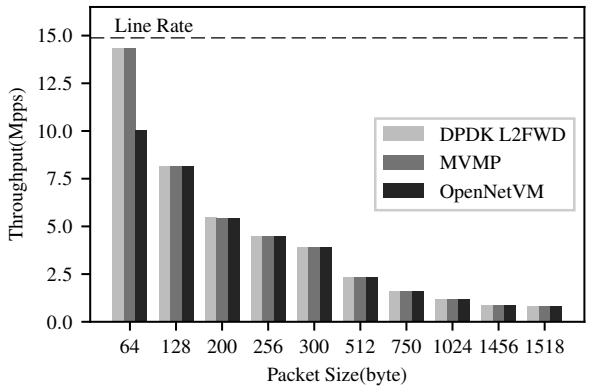



**Figure 5: Forwarding performance for different packet sizes, both using two logical CPU cores.**

30%. For 128-byte and larger packets, all approaches could saturate the 10GbE link.

**Packet Duplication:** MVMP facilitates traffic sharing by allowing multiple NFs to open the same virtual device, with a packet descriptor distributed to each NF's lockless rings. Since OpenNetVM does not support traffic duplication, only MVMP was evaluated for this case. In this experiment, the MVMP daemon uses two logical cores and assigns one logical core to each NF. As Figure 6(a) shows, as the number of NFs increases, the throughput decreases. The profiler tool shows that massive enqueuing and dequeuing in the lockless rings is responsible for the runtime overhead. After we increased the packet size to 256 bytes (Figure 6(b)), MVMP could saturate the bandwidth, which can be attributed to the fact that only the packet descriptor is copied.

### 4.2 Service Chain Performance

To evaluate the inter-NF communication performance, we built a service chain containing $k$ NFs and $k$-1 rootless devices. The first NF receives a packet from one virtual device and delivers it to the next NF with one rootless device, and so on, until the last NF sends the packet out. OpenNetVM was set to the same configuration with its auxiliary script. To make a fair comparison, we disabled its flow table. For SR-IOV, we created $k$ virtual NICs and had a process conduct a similar forward function by rewriting MAC addresses.

Figure 7 shows how the throughput for 64-byte packets forwarding changed as we adjusted the chain length. With a single NF, MVMP's and OpenNetVM's throughputs were consistent with Figure 5 (64-byte). As the number of NFs increased, MVMP was able to retain a stable speed. In contrast, the throughputs of OpenNetVM and SR-IOV dropped with more chained NFs. In particular, at five NFs, MVMP outperformed OpenNetVM and SR-IOV by as much as 3× and 7×, respectively. We believe that SR-IOV's inter-NF packet delivery performance decreased because of data copying costs and PCI communication.

## 5 RELATED WORK

E2 [10] is a framework for end-to-end orchestration of middleboxes. E2 steers traffic across appropriate NF instances with a predefined directed acyclic graph. However, the system is not flexible enough to deal with automatically triggered forwarding. mSwitch [7] is an implementation of software packet switching on a data plane. It is designed for the network performance of virtual machines, but not for multiple-NF service chains. ClickOS [9] has been proposed as a framework for NF deployment on a commodity platform; it focuses on efficient packet delivery through a hypervisor and virtual machines. OpenNetVM [16] was designed for high-perfomance service chains and deploys NFs in containers. The difference between OpenNetVM and MVMP lies in how they steer traffic. OpenNetVM uses a flow table and an additional thread to move packets through NFs. In contrast, MVMP provides a virtual device abstraction that allows NFs to define their own input and output and thus connect with each other. In a service chain that combines inline NFs and bypass NFs, the virtual device abstraction is more concise than a flow table.

## 6 CONCLUSION

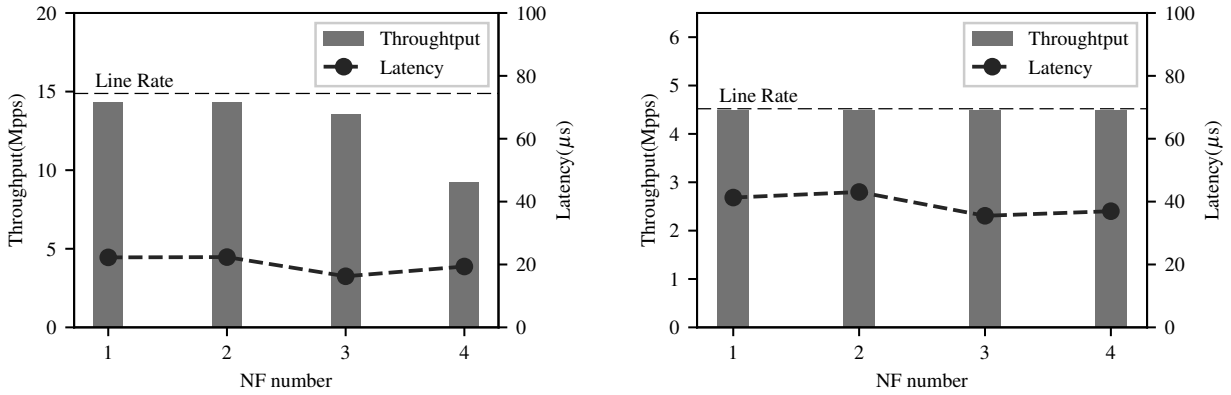
MVMP is a packet-processing platform for multiple middlebox networking. Our work focuses on deployment flexibility, inter-NF communication efficiency, and fault isolation. MVMP adopts a virtual device abstraction to decouple NFs from the underlying network topology. In addition, this abstraction makes inter-NF communication much easier. Based on Docker containers, DPDK, and a fine-grained shared-memory mechanism, MVMP can provide both proper fault isolation and high I/O performance. Our evaluation shows that MVMP can forward packets at 14.3Mpps (96% of the line rate) with one physical core (two logical cores). Our service chain experiment with five NFs shows that MVMP outperforms OpenNetVM by as much as 3×. Furthermore, while MVMP provides deployment flexibility and fault isolation, it retains ideal performance regarding traffic duplication and service chains.

## ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. This work is supported in part by National Key R&D Program of China (Grant NO. 2016YFB0801304), and National High Technology Research and Development Program of China (Grant No.2015AA016005).

## REFERENCES

[1] 2015. Data Plane Development Kit. (2015). https://en.wikipedia.org/wiki/Data_Plane_Development_Kit
[2] 2015. IXIA BreakingPoint. (2015). https://www.ixiacom.com/products/breakingpoint
[3] Luca Deri et al. 2004. Improving passive packet capture: Beyond device polling. In *Proceedings of SANE*, Vol. 2004. Amsterdam, Netherlands, 85–93.
[4] DPDK.org. 2014. Data plane development kit. (2014).
[5] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. 2015. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine* 53, 2




**(a) 64-byte packet** **(b) 256-byte packet**

**Figure 6: Virtual device sharing performance.**

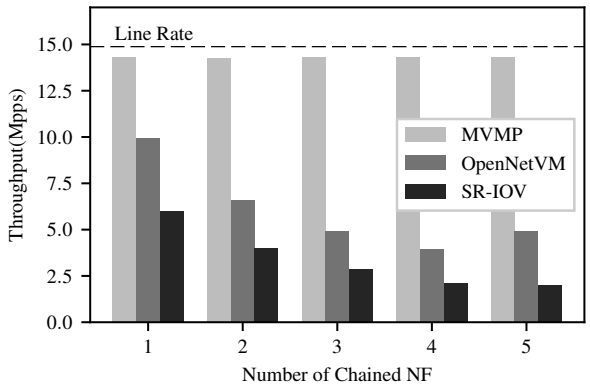



**Figure 7: Service chain performance, 64-byte packet.**

(2015), 90–97.

[6] Sangjin Han, Keon Jang, Kyoung Soo Park, and Sue Moon. 2010. PacketShader: a GPU-accelerated software router. In *ACM SIGCOMM 2010 Conference*. 195–206.

[7] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. 2015. mSwitch: a highly-scalable, modular software switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. ACM, 1.

[8] Mallik Mahalingam, D Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreeger, T Sridhar, Mike Bursell, and Chris Wright. 2014. *Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks*. Technical Report.

[9] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 459–473.

[10] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: a framework for NFV applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 121–136.

[11] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. 2015. The Design and Implementation of Open vSwitch.. In *NSDI*. 117–130.

[12] Luigi Rizzo. 2012. Netmap: a novel framework for fast packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)*. 101–112.

[13] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 95–103.

[14] Murari Sridharan, A Greenberg, N Venkataramiah, Y Wang, K Duda, I Ganga, G Lin, M Pearson, P Thaler, and C Tumuluri. 2011. NVGRE: Network virtualization using generic routing encapsulation. *IETF draft* (2011).

[15] Hancheng Wu, Da Li, and Michela Becchi. 2016. Compiler-assisted Workload Consolidation for Efficient Dynamic Parallelism on GPU. In *2016 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS)*.

[16] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. 2016. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM.