Isolation Mechanisms within the vSwitch of Cloud Computing Platform

Ye Yang

THÈSE DE DOCTORAT DE
SORBONNE UNIVERSITÉ

Spécialité
Informatique

Laboratoire d'Informatique de Paris 6
École Doctorale Informatique, Télécommunications et Électronique

Présentée par
Ye YANG

Pour obtenir le grade de
DOCTEUR de SORBONNE UNIVERSITÉ

Sujet de la thèse :

Isolation Mechanisms within the vSwitch of Cloud Computing Platform

soutenue le 27 Juin 2022

devant le jury composé de :

M. Serge FDIDA	Sorbonne Université	Directeur de thèse
M. Gaogang XIE	CNIC, CAS	Directeur de thèse
M. Kave SALAMATIAN	Université de Savoie	Rapporteurs
M. Ke XU	Tsinghua Université	Rapporteurs
M. Marcelo DIAS DE AMORIM	Sorbonne Université	Président du jury
M. Zhenyu LI	ICT, CAS	Examinateur
Mme. Thi-Mai-Trang NGUYEN	Sorbonne Université	Examinateur

Acknowledgements

During the few years of my Ph.D. period, I often felt confused, and I was always learning through trial and error. Due to the lack of logic and outstanding expression ability, it is not easy for me to sort out a very rigorous research logic. So I always got the "rejected" decisions for the poor expression in my submitted academic papers. Thanks to the help of professors, family, and friends, I have been able to establish my research area and make breakthroughs. I should be grateful to those who helped me.

Firstly, I want to thank my family. I want to thank my parents for providing me with the best possible studying and living conditions from I was a child, and encouraging me to go out of my small hometown to do research in a university and for a doctorate degree. They have paid a lot for me over the past twenty years. Then I would like to thank my girlfriend, Miss. Jiaqi Liang. Thank you for your love and encouragement during the low point of my life. You have changed my character that was too conservative, and taught me the speculative spirit in the real life.

Of course, I would like to thank Professor Serge Fdida and Gaogang Xie, my advisors. As the vice president of Sorbonne University, Professor Serge is usually involved in lots of official businesses, but he always gives full respect and much helps to students. Every time I send him an email to exchange manuscripts or ideas, Professor Serge will reply patiently no matter how busy he is. As my mentor in China, Professor Gaogang Xie can be regarded as a leader on the road of my scientific research career. He has established the correct values for me to do scientific research: starting from practical problems, considering the constraints in the actual environment, and seeking for the best solution. Only the solutions made in this circumstance are the most valuable scientific achievements that can be recognized by the industry.

Thanks to the engineers Xing Li and Yilong Lv from Alibaba Cloud. They provided me with many practical challenges and problems faced by the public cloud industry. In our cooperation, I also gradually enriched my understanding of cloud networking.

I cannot forget Emilie Mespoulhes, Zoe Jegu, and Patricia Zizzo. They helped me with remote registration and other formalities during the COVID-19 pandemic. It is a pity that I have not been able to come to Paris and Sorbonne University physically, and thus missed a

wonderful experience. The epidemic has affected the lives of countless, but I believe that the cold winter will eventually pass, and spring is just around the corner.

Abstract

As an important component of the cloud platform, virtual switch (vSwitch) is responsible for realizing the network connectivity between virtual machines (VMs) and external devices. In order to make great use of limited resources to forward packets with high performance, existing vSwitches have been widely adopted with sharing designs. For example, sharing hardware resources, data structure and processing procedures among VMs. However, these sharing designs destroy the isolation among VMs. In vSwitch, different VMs compete for shared resources and access memory without restriction, which makes them cannot be guaranteed with stable network Quality of Service (QoS) and also face the risk of data plane attacks along with illegal memory access.

In order to solve these performance, failure, and security issues caused by the lack of isolation, this thesis explores the isolation mechanisms from three aspects of hardware resources, software data structure, and I/O operations in vSwitch. In this way, the cloud service providers can really provide tenants with isolated, secure and stable virtual network environment. The main works and contributions of this thesis are as follows:

1) CPU-cycle isolation based network QoS method. To solve the problem that VMs compete for limited CPU resources in vSwitch to interfere with each other's network performance, this thesis proposes a novel network QoS method based on CPU-cycle isolation (C2QoS). C2QoS establishes the corresponding relationship between vSwitch forwarding capability and CPU consumption through a measurement-driven modeling method. Based on this model, C2QoS designs a CPU-cycle based token bucket mechanism, which provides bandwidth guarantee for VMs through isolating and restricting the I/O-dedicated CPU resources. Besides the token bucket mechanism, a hierarchical batch processing task scheduling mechanism is designed to provide differentiated latency according to priority. Through the realization of the proposed C2QoS method on the open-source OVS-DPDK platform, this thesis has carried out a sufficient experimental evaluation on it. The results show that, compared with the traditional packet/flow-based QoS method, the C2QoS method achieves VM network bandwidth guarantee by isolating

the competition of CPU resources, and at the same time reducing the additional VM network latency caused by competition by 80%.

2) Flow table isolation based data plane attack defense mechanism. Aiming at the problem of Denial of Service (DoS) attacks initiated by malicious tenants during the lookup process of the shared flow table, this thesis proposes a data plane attack defense mechanism based on flow table structure isolation (D-TSE). D-TSE uses VM as the unit to separate the flow table structure to achieve independent packet classification performance and failure isolation. In order to redirect packets to its dedicated flow table, D-TSE designs a lightweight pre-classification module to determine the attribution of each packet before the classification operation. To ensure the forwarding efficiency in the separated flow table structure, D-TSE designs a batch re-aggregation mechanism. Through the realization of the proposed D-TSE mechanism on the OVS-DPDK platform, this thesis has carried out sufficient experimental verification on it. The results show that D-TSE isolates the data structure and processing procedures belonging to different VMs in vSwitch at the cost of up to 5% performance degradation, thereby achieving the isolation of network failure and efficiently solving the risk of data plane DoS attacks.

3) Memory access isolation based virtualized network I/O (VNIO) mechanism. To solve the risk of illegal memory access caused by shared memory in existing VNIO mechanisms, this thesis proposes a VNIO mechanism based on memory access isolation (S2H). By analyzing the existing memory sharing models adopted in the VNIO mechanisms and their security risks, a secure memory sharing model is designed. Based on this model, S2H mechanism based on virtio standard is designed and implemented. In order to reduce the CPU usage and ensure scalability in the S2H mechanism, this thesis designs a "batch-grained" thread scheduling method. Through the realization of the S2H prototype system on the OVS-DPDK and QEMU/KVM platforms, this thesis has carried out a large number of experiments to verify its validity. The results show that S2H mechanism achieves the highest memory isolation and security in the software-based VNIO mechanisms at the cost of 2-9% increase in latency, while maintaining comparable performance and scalability as the widely adopted vHost-User solution.

Résumé

En tant que composant important de la plate-forme cloud, le commutateur virtuel (vSwitch) est responsable de la réalisation de la connectivité réseau entre les machines virtuelles et les périphériques externes. Afin de tirer le meilleur parti des ressources limitées pour transférer des paquets avec des performances élevées, les vSwitches existants ont été largement adopté des principes de conception partagés. Par exemple, partager les ressources matérielles, la structure des données et les procédures de traitement entre les machines virtuelles. Cependant, ces conceptions de partage détruisent l'isolement entre les machines virtuelles. Dans vSwitch, différentes machines virtuelles se disputent les ressources partagées et accèdent à la mémoire sans restriction, cela les rend incapables de garantir une qualité de service (QoS) réseau stable, tout en faisant face au risque d'attaques de plans de données et d'accès illégaux à la mémoire.

Afin de résoudre ces problèmes de performance, de défaillance et de sécurité causés par le manque d'isolation, cette thèse explore les mécanismes d'isolation de trois aspects des ressources matérielles, de la structure des données logicielles et des opérations d'E/S dans vSwitch. De cette façon, les fournisseurs de services cloud peuvent vraiment fournir aux locataires un environnement de réseau virtuel isolé, sécurisé et stable. Les principaux travaux et contributions de cette thèse sont les suivants:

1) Méthode QoS réseau basée sur l'isolation du cycle CPU. Pour résoudre le problème selon lequel les machines virtuelles se disputent des ressources CPU limitées dans vSwitch pour interférer les performances réseau de l'autre, cette thèse propose une nouvelle méthode QoS réseau basée sur l'isolation du cycle CPU (C2QoS). C2QoS établit la correspondance entre la capacité de transfert vSwitch et la consommation CPU grâce à une approche de modélisation basée sur la mesure. Sur la base de ce modèle, C2QoS conçoit un mécanisme de seau à jetons basé sur le cycle CPU, qui fournit une garantie de bande passante pour les machines virtuelles en isolant et en limitant les ressources CPU dédiées aux E/S. Outre le mécanisme de seau à jetons, un mécanisme de planification hiérarchique des tâches de traitement par lots est conçu pour fournir une latence différenciée en fonction de la priorité. A travers la réalisation de la méthode C2QoS proposée sur la plateforme open source OVS-DPDK, cette thèse a réalisé une évaluation expérimentale suffisante sur celle-ci. Les résultats montrent que, par rapport aux méthodes QoS traditionnelles basées sur les paquets/flux, la méthode C2QoS garantit la bande

passante du réseau VM en isolant les conflits de ressources CPU, tout en réduisant la latence supplémentaire de 80%.

2) Mécanisme de défense contre les attaques du plan de données basé sur l'isolement de la table de flux. Visant le problème des attaques par déni de service (DoS) initiées par des locataires malveillants lors du processus de recherche de la table de flux partagée, cette thèse propose une défense contre les attaques du plan de données mécanisme basé sur l'isolation de structure de table de flux (D-TSE). D-TSE utilise VM comme unité pour séparer la structure de la table de flux afin d'obtenir des performances de classification de paquets indépendantes et une isolation des pannes. Afin de rediriger les paquets vers sa table de flux dédiée, D-TSE conçoit un module léger de pré-classification pour déterminer l'attribution de chaque paquet avant l'opération de classification. Pour garantir l'efficacité du transfert dans la structure de table de flux séparée, D-TSE conçoit un mécanisme de réagrégation par lots. En implémentant le mécanisme D-TSE sur la plate-forme OVS-DPDK, cet article l'a entièrement vérifié expérimentalement. Les résultats montrent que D-TSE isole la structure des données et les procédures de traitement appartenant à différentes machines virtuelles dans vSwitch au prix d'une dégradation des performances allant jusqu'à 5%, réalisant ainsi l'isolation des pannes de réseau et résolvant efficacement le risque d'attaques DoS du plan de données.

3) Mécanisme d'E/S réseau virtualisé (VNIO) basé sur l'isolation de l'accès mémoire. Pour résoudre le risque d'accès illégal à la mémoire causé par la mémoire partagée dans les mécanismes VNIO existants, cette thèse propose un mécanisme VNIO basé sur l'isolement de l'accès mémoire (S2H). En analysant le modèle de partage de mémoire adopté par le mécanisme VNIO existant et ses risques de sécurité, cet article conçoit un modèle de partage de mémoire sécurisé. Sur la base de ce modèle, un mécanisme S2H basé sur la norme virtio est conçu et mis en œuvre. Afin de réduire l'utilisation du processeur et d'assurer l'évolutivité du mécanisme S2H, cette thèse conçoit une méthode d'ordonnancement de threads "batch-grained". A travers la réalisation du système prototype S2H sur les plateformes OVS-DPDK et QEMU/KVM, cette thèse a réalisé un grand nombre d'expérimentations pour vérifier sa validation. Les résultats montrent que le mécanisme S2H atteint l'isolation et la sécurité de la mémoire les plus élevées dans les mécanismes VNIO basés sur le logiciel au prix d'une augmentation de 2 à 9% de la latence, tout en maintenant des performances et une évolutivité comparables à celles de la solution vHost-User largement adoptée.

Contents

List of Figures

List of Tables

Publications

Y. Yang, H. Jiang, Y. Wu, C. Han, Y. Lv, X. Li, B. Yang, S. Fdida, and G. Xie, “C2QoS: Network QoS guarantee in vSwitch through CPU-cycle management”, Journal of Systems Architecture, vol. 116, p. 102148, 2021.

Y. Yang, H. Jiang, G. Zhang, X. Wang, Y. Lv, X. Li, S. Fdida, and G. Xie, “S2H: Hypervisor as a setter within virtualized network I/O for VM isolation on cloud platform”, Computer Networks, vol. 201, p. 108577, 2021.

Chapter 1

Introduction

1.1 The overview of vSwitch on the cloud platform

Cloud computing, known for its high cost-effectiveness and flexibility, has been popular and widely deployed, since it was proposed in 2005. After several years of development, lots of emerging technologies and scenarios including data centers, operator networks, edge computing, and even 5G have now been integrated into cloud platforms and architectures. More and more enterprises and individuals are also deploying their services by purchasing virtual machines (VMs) on cloud platforms[1–4]. Thanks to virtualization technology, these VMs running different services can enjoy an independent and flexible running environment on the same physical server. That is achieved by the virtualization layer, which is shown in Fig. 1.1. The virtualization layer between the hardware and the VMs, provides VMs with the abstraction of physical hardware, which enables VMs to reasonably use these physical resources on demand.

Generally, the hypervisor layer consists of 4 kinds of virtualization technologies, i.e. CPU virtualization, memory virtualization, storage virtualization, and network virtualization. The first three are included in the hypervisor program of the corresponding virtualization platform, which provides the virtualization of real physical hardware resources, such as QEMU/KVM[5, 6], Xen[7], etc. However, the network virtualization is different. On a host server, the VM's network resource is not a real physical resource, but a "virtual" resource realized by a third-party vSwitch through software packet forwarding. The VM's maximum network bandwidth and latency are also determined by the performance of vSwitch. Therefore, it can be said that the network virtualization is provided by vSwitch independent from the virtualization platform[8, 9].

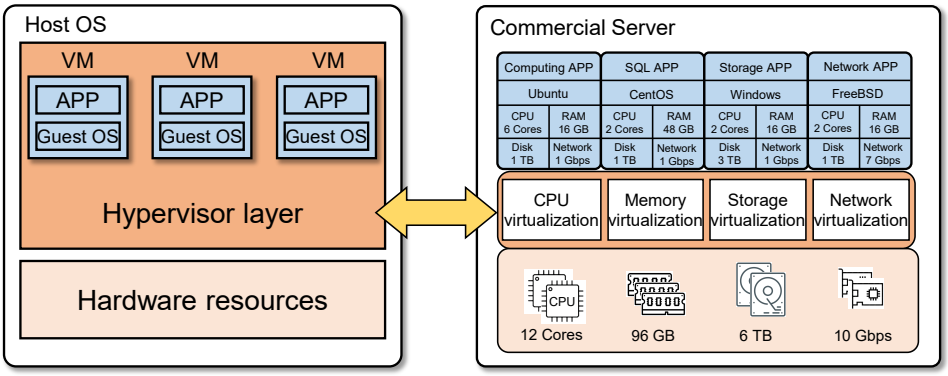
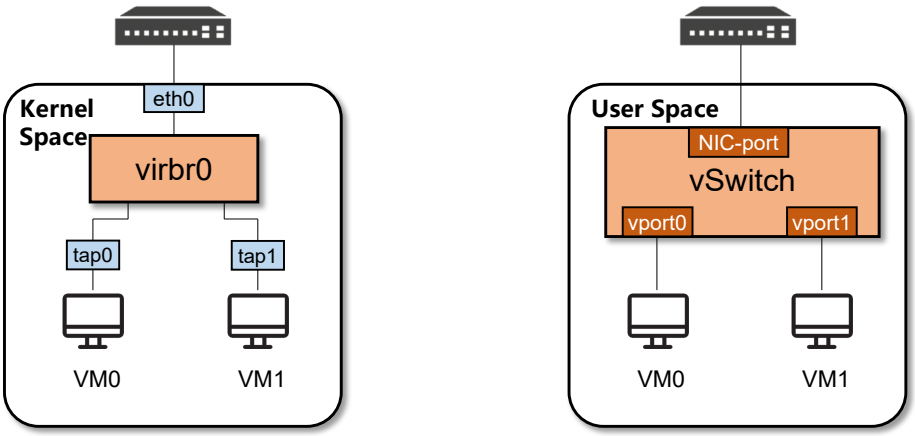


Figure 1.1 The cloud platform infrastructure

The idea of leveraging vSwitch to virtualize the network is originated from Bridge[10] in the Linux kernel. As shown in Fig. 1.2(a), Linux Bridge is a virtual network device that works on a layer 2 network. It can bind other network devices as slave devices, and virtualize the slave devices into "ports". Similar to a switch in the physical network, when a device (such as a network interface controller (NIC) device “eth”, or a virtual “tap” device) is bound to the Linux Bridge as a port, it can use this Bridge to switch packets. However, before the packet is switched and forwarded in Linux Bridge, it needs to be copied many times in the kernel space. That leads to low performance. On the other hand, Bridge can only implement basic network switching functions, but cannot manage, configure, and monitor complex virtual networks in a cloud environment. Therefore, the high-performance and powerful vSwitch came into being.



(a) Linux Bridge (b) vSwitch

Figure 1.2 The comparison of Linux Bridge and vSwitch.

The structure of the vSwitch is shown in Fig. 1.2(b). To pursue higher performance, the existing vSwitches are transferred from kernel space to user space, which can reduce the memory

copying times. Another benefit is to make it easier to leverage various high-performance user-space packet processing components for acceleration, such as Intel Data Plane Development Kit (DPDK)[11], etc. In terms of flexibility, the biggest difference between vSwitch and Linux Bridge is that the vSwitch contains the whole network forwarding and network I/O logic, that is, all the processes from device driver to packet forwarding are all included in the vSwitch program. In this way, the vSwitch has the privilege to perform fine-grained traffic processing, configuration, and monitoring at different levels (e.g. per-tenant, per-port, per-flow, and even per packet), which greatly enriches the cloud service provider's ability to manage virtual networks.

For each VM deployed on the cloud platform, the vSwitch can provide it with basic network functions in three aspects: virtualized network I/O (VNIO), packet classification, and external network connection. Take the VM sending packets as an example: when a packet is sent from the VM, the vSwitch will first call the interface function of the VNIO to realize packet copying on the corresponding virtual port (see "vport" in Fig. 1.2(b)); when the packet is copied from the VM memory to the vSwitch's packet buffer, the packet classification algorithm in vSwitch will parse the packet header and search the destination port according to the quintuple information; on the destination port, vSwitch sends the packet to external network by calling the sending callback function. The whole process takes place in user-space and is carried out in batch processing mode, which makes it efficient. So it can be said that vSwitch has already realized the vision of providing a high-performance and flexible virtual network for tenants on the cloud.

However, with the improvement of vSwitch performance, the number of VMs it can support has also increased significantly, and that introduces isolation issues. In the evolution of vSwitch, the developers have always regarded flexibility as their primary goal, and therefore integrate a large number of network functions and components into vSwitch. At the same time, in order to maximize the use of limited resources and provide efficient forwarding capacity, designs such as sharing resources and data structures among tenants are adopted. These designs have made the vSwitch gradually become a highly privileged user-space process with resource sharing and intensive I/O operations. In a high-density deployed cloud platform, it is difficult to guarantee the isolation of tenants' network performance, network failure, and even security. Therefore, the problem of isolation in vSwitch becomes an urgent challenge for cloud service providers (CSPs) currently.

1.2 Isolation challenges in vSwitch

For the isolation problem within the vSwitch of cloud platform, the key point is to research how to combine the architecture of vSwitch to design mechanisms for guaranteeing VM isolation in network performance, network failures, and security. Specifically, it includes the following aspects: isolating and guaranteeing tenant's network Quality of Service (QoS) and Service Level Agreements (SLA); isolating forwarding failure and defending against data plane Denial of Service (DoS) attacks; isolating and protecting the VM memory from illegal access by attackers during the VNIO operations. However, the existing works lack a systematic analysis and theoretical basis for these issues, which therefore ignores the competition resources, the sharing of data structures, and the highly privileged I/O operations. As a result, they all cannot achieve good results.

1.2.1 Network performance isolation

Network performance isolation in vSwitch requires that the VM's network SLA performance is stable, independent, and not affected by other tenants. That is achieved through QoS methods. The existing QoS methods used in vSwitch are inherited from hardware switches, including two major types of mechanisms: token bucket mechanisms[12–15] and fair queuing mechanisms[16–20]. The core idea of these two mechanisms is to forward the packets at a certain rate or interval, so as to ensure the network bandwidth and latency requirements. However, the prerequisite for these mechanisms is that the processing capacity of the network forwarding device is constant, and the processing time and resource consumption per packet are also constant. Unfortunately, this premise is common in hardware switches, but it does not apply in software-implemented vSwitches.

The latest works proved that in the vSwitch, the forwarding capacity of the CPU core processing engine will change with the different network traffic loads[21–23]. Therefore, a tenant may compete for limited CPU physical resources in the software vSwitch by constructing traffic with extreme characteristics, thereby disturbing the network experience of other tenants. In order to avoid the problem of VM network performance interference due to the lack of CPU resource isolation, these works only add a CPU limiting module besides the traditional hardware switch based QoS method. The disadvantages are obvious: on the one hand, these methods cannot accurately describe the relationship between variable network performance and CPU resource consumption; on the other hand, they will increase the overhead in vSwitch for that multiple modules are added to the data path.

However, to redesign the QoS methods for software vSwitch according to its architecture and isolation demands on hardware resources, there are three main challenges:

1) The forwarding capability of the vSwitch is variable, making it difficult to establish a resource consumption model corresponding to the network forwarding performance. In vSwitch, the processing engine is a general-purpose CPU core. Compared to the powerful hardware pipelines in hardware switch, that has a constant packet forwarding rate, the limited CPU core in vSwitch has different forwarding capacity under different traffic characteristics. In Google Cloud's experiments, one CPU core can forward 1518-byte packets to 10 Gbps, but can only reach 1.2 Gbps with 64-byte packets. It is the variable processing capacity that causes the traditional QoS method to fail in vSwitch, because the time used for forwarding each packet is different, and the CPU resources consumed are also different.

2) The fine-grained CPU resource allocation and limitation. On cloud servers, most of the CPU resources need to be allocated to VMs in order to pursue higher profits. Therefore, the CPU cores that can be allocated to vSwitch for packet forwarding are limited. Taking Google Cloud as an example, only 2 CPU cores are allocated to the Google's vSwitch to forward traffic for all VMs on one physical server. However, in the field of software forwarding, existing CPU resource allocation and isolation strategies are all based on CPU core granularity, which is unrealistic in vSwitches with limited CPU cores. On the other hand, how to use CPU resource allocation to accurately limit the VM's packet rate also faces challenges in terms of lightweight and implementation.

3) The batch processing and polling running modes pose challenges to traffic scheduling and latency guarantee. For efficiency, batch processing and polling running modes are widely used in the vSwitch to achieve the maximum throughput. The running mode of each working CPU core in vSwitch is as follows: receives and forwards a batch of (e.g. 32) packets on a VM's port, and then goes to the next port to perform the same operations. As the density of VMs deployed on one server continues to increase, these VMs inevitably compete for a limited number of CPU cores in terms of timing, resulting in increased network latency. The traditional packet and traffic scheduling methods both work during the process of packet forwarding, and do not take into account that the packet forwarding tasks compete for CPU core in terms of timing. Therefore, it cannot solve the indiscriminate high latency that forwarding tasks wait for the CPU core to process.

Through the above analysis, it can be seen that the existing network QoS methods in vSwitch are inherited from the hardware switch, and do not take into account the competition of the traffic forwarding for CPU resource in terms of utilization and timing, resulting in poor isolation

and cannot guarantee the tenant SLA performance. Therefore, it is necessary to redesign a new QoS method based on hardware resource isolation according to the vSwitch's architecture. This research aims to address the three challenges in proposing the new QoS method for isolating VM network performance.

1.2.2 Network failure isolation

Network failures in vSwitch mainly occur in the flow table lookup module. On the data plane of the virtual network, there is a kind of DoS attack, using the flow table update mechanism under Software Defined Network (SDN) to create chaos and reduce the lookup performance. This DoS attack will then lead to the forwarding failure in the forwarding device, and cause network downtime problems to all tenants. Regardless of whether the attack is initiated by an external device or an internal tenant, the failure of the forwarding device will affect all tenants who use the device for packet forwarding. As early as the birth of SDN, some works have warned that a kind of DoS attack leveraging the flow table update feature may occur in all SDN devices[24, 25]. But most of them require a large number of packets to achieve, so it is easy to detect and prevent through big flow detect mechanisms, thereby having little impact on the real network environment[26, 27].

However, some recent works have found that the lack of isolation in the vSwitch's flow table can be used to bypass the detection mechanism and implement DoS attacks by carefully constructing some low-rate traffic[28, 29]. Since all VMs share the same flow table structure in the vSwitch to classify and forward packets, a malicious tenant can cause the lookup frequently miss in the shared flow table by constructing special low-rate traffic, thereby triggering The SDN controller issues a large number of meaningless rules and inserts them into the shared flow table. These frequently inserted meaningless rules will greatly increase the time and space complexity of flow table lookup, and the DoS attack formed. During the attack, all VMs' packet lookup processes will become extremely slow because of this common "failure point", and the forwarding performance of the entire vSwitch can be reduced to less than 20% at most. Existing defense mechanisms against this attack focus on limiting the frequency of traffic "pulling down" new rules and improving the packet classification algorithms[28, 30]. Unfortunately, these mechanisms can only reduce the effectiveness of DoS attacks, but cannot completely isolate and prevent such attacks. What's worse, the forwarding performance of some normal services will also be affected by these restrictions.

The fundamental reason for the DoS attack on the data plane is the lack of isolation in vSwitch's flow table structure, so this research starts from the idea of isolating the flow table structure to

reduce attack risks and isolate the device failure. For implementation, separating the shared flow table in the vSwitch faces the following two challenges:

1) Break the paradox of needing to predict the packet's owner before classifying it. After separating the shared flow table for each tenant in vSwitch, a problem occurred that before the packet needs to be classified, it should be first determined its owner, so as to redirect the packet to the corresponding flow table structure for classification. This problem is particularly serious in the direction of NIC to VM, because the packets received from NIC could belong to any VMs on the server. Thus, an efficient pre-classification mechanism needs to be carefully designed.

2) Reduce the overhead brought by the separated flow table architecture. When the shared flow table in the vSwitch is split into multiple private flow tables of different tenants, the packet processing procedure also changes accordingly. Due to the existence of the batch processing mechanism, a batch of packets received on one port may be redirected into different flow tables for subsequent lookup and forwarding operations, which will greatly reduce the relevance of context processing. The most typical problem is that the cache locality is invalid, thereby reducing the forwarding efficiency.

Based on the above analysis, to isolate the common failure and prevent the DoS attack on the data plane of vSwitch, it is necessary to add the flow table structure isolation design to the vSwitch. This research focuses on solving the above two challenges, and proposes an efficient packet processing procedure under the separated flow table architecture, as well as some forwarding performance optimization mechanisms.

1.2.3 Memory security isolation

Illegal memory access in the virtualization environment is mostly caused by I/O operations, especially the VNIO accelerated by shared memory. The role of VNIO is to transfer packets between the VM memory and the vSwitch's packet buffer. Since these two pieces of memory belong to different processes on the host (hypervisor and vSwitch), it requires an efficient inter-process communication mechanism. The existing VNIO mechanisms widely adopt the design of shared memory to improve the data path performance of vSwitch. In terms of memory sharing, the existing solutions adopt two types of memory sharing models: the VM memory is shared to vSwitch, and vSwitch realizes the packet copying workload[31–33]; the vSwitch's packet buffer is shared with all the VMs, and the VM directly reads and writes the packets in the buffer[34–38].

These memory sharing models bring high-performance data paths for vSwitch, but also break the memory isolation originally provided by the virtualization platform and bring serious security risks. The memory sharing models introduce a new attack surface for attackers, in which malicious tenants can initiate illegal memory access. Through shared memory, an attacker can read or write the memory of other VMs without restriction. Taking the first memory sharing model as an example, malicious tenants can construct packets to trigger the vulnerabilities in vSwitch. After taking control of the user-space vSwitch, they can implement "legal" memory access and even attack to other VMs.

The virtualization open source community has already been aware of this risk, and researchers from RedHat also reported a Direct Memory Access (DMA) attack based on shared memory[39, 40]. However, the existing solutions are all simple reinforcement works, which are dedicated to restricting memory access behavior based on insecure memory sharing models. For example, the address translation process of memory access is restricted to prevent illegal I /O memory access. However, these reinforcement measures greatly reduce the performance due to frequent legality checks and inter-process communication. For example, the price paid by the community's vIOMMU solution is that the VNIO performance is reduced to 20%[41–43]. In addition, other memory protection mechanisms, such as the hardware-based Intel Software Guard eXtensions (SGX) mechanism, cannot be used in VNIO for the same performance reason[44–46].

Since simple reinforcement solutions cannot solve security risks while ensuring performance, this research attempts to analyze the root cause of these attacks. As the VM memory isolation is broken by the insecure memory sharing model, the best solution should start from the shared memory design, and to explore safe but efficient memory sharing model in VNIO design. Achieving this goal will face three challenges:

1) Achieve the trade-off between security and performance. The introduction of shared memory is to reduce memory copying times for accelerating VNIO, and improve network performance. Therefore, if the memory sharing model should be redesigned to enhance isolation, it cannot introduce new memory copying times and communication overhead.

2) The compatibility support for existing cloud infrastructure. The VMs running on the cloud have strong compatibility requirements, such as allowing tenants to use various system images, various network drivers, etc. Therefore, the design of VNIO mechanism cannot seek support for modifying components inside the VM. Furthermore, all modifications to the architecture also need to be tenant-agnostic and compatible with existing component interaction methods.

3) Reduce additional CPU usage to meet scalability. Since the vSwitch on the cloud can only use limited CPU cores for packet forwarding, the design of VNIO mechanism also needs to

take into account the CPU usage. In order to ensure memory isolation, the shared memory and data paths are required to be redesigned, which may increase the number of threads for copying packets. So innovations in scheduling methods are needed to meet CPU quota requirements.

To summarize, in order to achieve the trade-off between performance and security isolation in the VNIO mechanism, this research aims to redesign more secure and efficient memory sharing along with the data paths. After solving these three challenges, this thesis will propose a safe and efficient VNIO mechanism for multi-tenant cloud platforms.

1.3 Research content and main contributions

This thesis focuses on the isolation challenges on tenant network performance, network failure, and memory security within the vSwitch of cloud computing platforms. The specific research content is shown in Fig. 1.3: firstly, in order to isolate VM network performance, a network QoS method based on CPU-cycle isolation (C2QoS) is proposed, which ensures that tenants enjoy independent network experience[47, 48]; secondly, to prevent DoS attacks on the data plane, a defense mechanism based on flow table structure isolation (D-TSE) is proposed, which isolates the possible common failure point in vSwitch and effectively reduces the risks of data plane attack; finally, to solve the illegal access to VM memory in the existing VNIO mechanisms, a memory access isolation based VNIO mechanism (S2H) is proposed, which take into account both the performance and security[49, 50]. The above research content starts from the urgent needs of the cloud platforms, provides an isolated, secure, and stable virtual network environment for VMs, and realizes the transition from “providing services to tenants” to “providing stable services to tenants”.

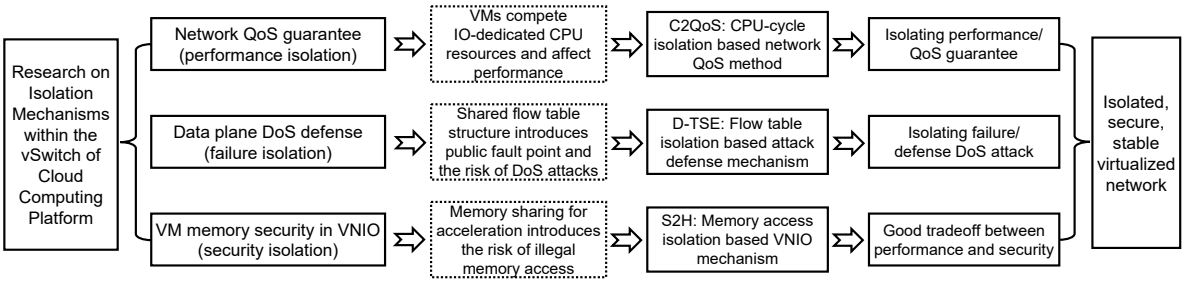


Figure 1.3 The research content

1.3.1 Research content

The specific research content of the thesis is as follows:

(1) Research on network QoS method based on CPU-cycle isolation

The existing QoS methods in vSwitch are inherited from hardware switches. These packet/flow-based methods ignore the competition for hardware resources and variable processing capability of vSwitch, so they cannot achieve expected results. In order to provide isolation on physical resources, redesigning a new QoS method faces three challenges: the CPU consumption of packet forwarding is affected by many factors, and it is difficult to model accurately; the CPU resources used by the cloud platform for packet forwarding are limited, so fine-grained isolation and allocation mechanisms of CPU resources are required; the batch processing and polling running modes in the vSwitch bring difficulties to the scheduling and latency guarantee of packets. This part of the work aims to solve these three challenges, based on the idea of CPU resource isolation, to design a method for ensuring VM network QoS.

This thesis proposes a network QoS method based on CPU-cycle isolation, named C2QoS. Specifically, C2QoS uses a measurement-driven method to describe the influence of VM traffic characteristics and deployment configurations on the IO-dedicated CPU consumption, and establishes the performance-CPU model. Based on this model, C2QoS designs a CPU-cycle based token bucket mechanism to accurately guarantee the tenant's network bandwidth, through isolating the competition on CPU utilization. In order to solve the competition in timing, C2QoS designs a hierarchical batch scheduling mechanism to schedule batch forwarding tasks on limited CPU cores according to their priorities, so as to provide differentiated latency.

By implementing the proposed C2QoS method on the open-source DPDK-accelerated Open vSwitch (OVS-DPDK) platform, this thesis fully verified its effectiveness. The experiments are conducted with two kinds of traffic: tester generating traffic and real services generating traffic. The experimental results show that, compared with the traditional packet/flow-based QoS method, the C2QoS method can strictly guarantee the VM network bandwidth, and at the same time reduce up to 80% of the additional network latency caused by competition.

(2) Research on data plane attack defense mechanism based on flow table isolation

Existing data plane attack defense mechanisms only focus on how to reduce the effect of data plane DoS attacks by detecting big flows and improving lookup algorithms. However, the newly proposed Tuple Space Explosion (TSE) attack has been achieved by constructing low-rate traffic to bypass the detection mechanism. Attack traffic will "pull down" a large number of

useless rules and insert them into the shared flow table in vSwitch, thereby increasing the lookup complexity and creating common failure points to interfere with other tenants. In order to solve this problem, this part of the work aims to isolate the flow table in vSwitch to prevent DoS attacks, and focuses on solving the paradox that packets need to be determined owners before looking up in the flow table.

This thesis thus proposes a data plane attack defense mechanism based on flow table isolation, named D-TSE. Specifically, D-TSE designs a separated multi-level flow table architecture based on VM granularity. The classification of each VM's packets will be performed in its private flow table, thus isolating lookup performance and possible failure points. In order to solve the paradox illustrated before, D-TSE adds a lightweight PRECLS module, between the packet receiving and flow table classification steps, to distinguish packet's owner VM through IP address and VxLAN identifier. Then, PRECLS redirects the packet to its corresponding flow table for subsequent lookup. Finally, in order to ensure the efficiency of packet processing under the separated flow table architecture, D-TSE designs a batch re-aggregation mechanism, which re-aggregates the redirected packets according to their destination VMs, and then forwarded in batches.

This thesis implements the proposed D-TSE mechanism on the OVS-DPDK platform, replacing the original shared multi-level flow table with a separated multi-level flow table architecture. The experimental results show that the D-TSE mechanism isolates the VM's flow table lookup performance and common failure points at the cost of a 5% performance drop, and effectively reduces the risk of DoS attacks initiated through the data plane flow table.

(3) Research on VNIO mechanism based on memory access isolation

As a bridge to transfer packets between the host and the VM, existing VNIO mechanism adopts an insecure memory sharing model to reduce memory copying times for higher performance, thereby introducing risks of memory illegal access. The reinforcement solution from the community achieves security through address legitimacy detection, and pays an expensive price of reducing performance by 80%, so it cannot be used in a production environment. In order to ensure the memory isolation of VMs without reducing performance, this part of the work attempts to design a safe memory sharing model without compromising performance, compatibility and CPU efficiency.

This thesis proposes a VNIO mechanism based on memory access isolation, named S2H. Specifically, this thesis firstly models the shared memory used in existing VNIO mechanisms, and analyzes their security. Based on the analysis, a secure memory sharing model is proposed to build VNIO data paths. According to the model, S2H, a safe and efficient VNIO mechanism

based on the virtio standard, is designed and implemented. S2H can support stock VMs, and achieves good compatibility with cloud infrastructure. In order to solve the problem of CPU efficiency and concurrent thread competition in S2H, this thesis designs a batch-grained thread scheduling method to save CPU resources and avoid deadlock when concurrent threads operate shared memory, which increases the scalability.

This thesis implements a prototype system of the S2H mechanism on the virtualization platform built by open-source QEMU/KVM and OVS-DPDK. The experimental results proved that the S2H mechanism achieves the highest isolation and security in the software VNIO mechanisms at the cost of only a 2–9% increase in latency, while obtaining the comparable performance and scalability to the most widely deployed vHost-User mechanism.

1.3.2 Main contributions

With all these works completed, the main contributions and innovations of this thesis are as follows:

- To isolate VM network performance, this thesis proposes C2QoS, a network QoS method based on CPU-cycle isolation, and implements it on the OVS-DPDK platform. Compared with the existing QoS methods, the C2QoS method can more strictly isolate and guarantee the VM network bandwidth, while reducing 80% of the additional latency caused by competition.
- To isolate VM network failure, this thesis proposes D-TSE, a data plane attack defense mechanism based on flow table structure isolation, and implements it in the OVS-DPDK platform. The experimental results show that the D-TSE mechanism isolates the common network forwarding failures at the cost of up to 5% performance degradation, and effectively prevents the data plane DoS attacks based on the shared flow table structure.
- To isolate VM memory security, this thesis proposes S2H, a VNIO mechanism based on memory access isolation, and implements its prototype system on OVS-DPDK and QEMU/KVM platform. Experimental results show that compared with the currently widely used vHost-User mechanism, S2H maintains considerable throughput, but increases 2–9% latency, while ensuring the VM memory isolation.

1.4 Thesis outline

The rest of this thesis is divided into five chapters.

The Chapter 2 summarizes the related works and background knowledge of vSwitch isolation in cloud platforms. The Chapter 3 introduces the design of C2QoS and its guarantee on VM network performance. The Chapter 4 D-TSE design to defense data plane DoS attack. Chapter 5 designs and implements the S2H mechanism with a secure memory sharing model to achieve a good trade-off between performance and security. Chapter 6 concludes the thesis.

Chapter 2

Background and motivation

Cloud computing, first proposed in 2006, has become a deployment paradigm for massive internet applications and services for its cost-effectiveness, flexibility, and scalability. After more than ten years of evolution, thanks to the virtualization technology, cloud computing has become an indispensable part of business and personal life. For example, the latest 5G communication technology adopts cloud infrastructure to deploy core network services, and Microsoft has also launched the Azure platform to help operators for deploying and managing physical networks through flexible management methods of cloud networks[51].

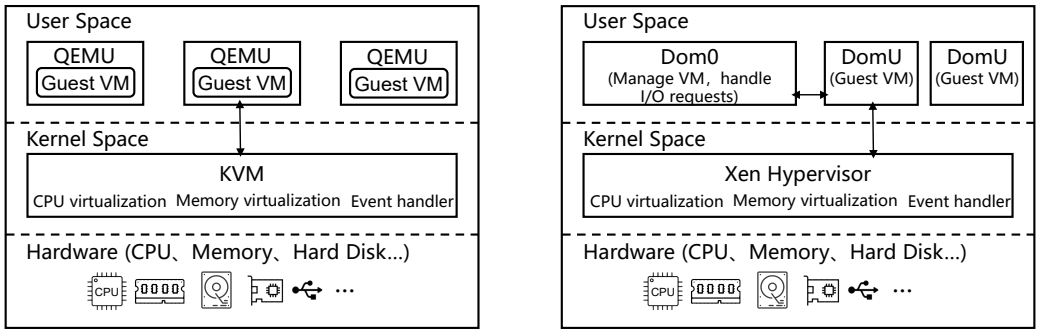
This thesis focuses on the vSwitch technology on the cloud platform. As the lowest-level switching and forwarding device in the cloud network, vSwitch not only needs to break the resource silos to provide high-performance interconnected networks for cloud services, but also keep tenants' isolation requirements in terms of performance, security, etc. This chapter will introduce the related technologies of vSwitches, then discuss the isolation that tenants require in vSwitch and the existing works in these areas.

2.1 The background of vSwitch

2.1.1 The virtualization technology

As the cornerstone of cloud computing, virtualization technology allows different tenants to be deployed on the same physical server in the form of VMs, thereby reusing the hardware resources on the server while enjoying an isolated operating environment. Technically, this is

achieved through a virtualization layer, also known as a Virtual Machine Monitor (VMM) or hypervisor. The hypervisor realizes the abstraction of hardware resources, that is, realizes the virtualization of CPU, memory, and even I/O handler to provide corresponding interfaces for the normal operation of VMs.



(a) QEMU/KVM (b) Xen

Figure 2.1 The mainstream virtualization architectures

Existing mainstream virtualization platforms are shown in Fig. 2.1. In the QEMU/KVM virtualization[5, 6] and Xen virtualization[7], the hypervisor includes parts in both user space and kernel space. Among them, the part in the kernel space runs at a higher privilege level (KVM in Fig. 2.1(a) and Xen hypervisor in Fig. 2.1(b)), and is responsible for completing CPU virtualization and memory virtualization, etc. The part in the user space is the main process of each VM (the QEMU process in Fig. 2.1(a) and the DomU in Fig. 2.1(b)), providing support from the host OS for the operation of VM. It is worth noting the difference in I/O request processing between the two virtualization mechanisms: In QEMU/KVM virtualization, each QEMU process is solely responsible for device emulation and I/O request processing of one VM within it, while Xen virtualization launches a special Dom0 VM to implement I/O requests and network communication for all the VMs. Therefore, the Xen virtualization platform itself consumes much more physical resources than QEMU/KVM platform. The CSPs also tend to choose the QEMU/KVM virtualization platform to build their cloud infrastructure with minimal resource cost, thereby reserving as many resources as possible for tenants[52].

Thanks to the virtualization platform, tenants in VMs can enjoy almost bare-metal performance in isolated environments. For the tenant, what he controls is only the part inside the VM Operation System (OS), but all the CPU instructions, I/O requests, etc. need to rely on the hypervisor to execute safely on the hardware. From the perspective of the host that VMs running on, each VM is actually a user-space hypervisor process, so it enjoys process-level operational independence and resource isolation.

However, with the density of VMs deployed on each server is increasing, how to realize the network connection and network management for the high-density deployed VMs is a big challenge for CSPs. That requires advanced vSwitch technology.

2.1.2 The vSwitch technology

To realize the network traffic forwarding and bridge the VM's communication with the outside devices, a series of difficulties need to be overcome. In terms of architecture, since the VMs run on a high software abstract, a packet in the VM needs to go through the virtualization layer, the host OS protocol stack, and the NIC before it can be sent out. During this period, components that can monitor and set forwarding rules are required. To solve this problem, the vSwitch has undergone the following stages of evolution.

1) TAP/TUN device communication. TAP/TUN[53] is an important function provided by the Linux system for user-space programs to realize network connection. In essence, they are all virtual network devices. TUN is located in the 3rd layer, simulating network layer devices, while TAP devices run at Layer 2. They all can be used to simulate the virtual NIC function for the VMs. When the VM needs to send packets out, it can write packets to TUN/TAP devices through the hypervisor, and the host OS kernel can receive packets through these devices. The situation in the opposite direction is similar.

2) The Linux Bridge implements a simple virtual switching function. After we have got the virtual devices on the host, a component that provides lookup and switching for packets from the virtual devices is necessary. This component is Bridge[10] in the Linux kernel. Linux Bridge is similar to a switch in the physical network. All network devices connected to it are virtualized into a port. Packets received from each port are forwarded in the Bridge through the mapping relationship between MAC addresses and ports. For example, if both the TAP device and the NIC device are added to the same Bridge, the VM can use this Bridge to realize packet switching and communicate with the outside devices. However, the use of TAP devices and Linux Bridge has two defects. The first one is that the packets will be copied multiple times in the kernel space, resulting in low performance. Another drawback is that the Linux Bridge cannot monitor and manage traffic. In this way, the VM traffic monitoring and management tasks fall on the external physical switches. Since all the devices bound to the Linux Bridge will lose their IP addresses, the traffic of different VMs cannot be distinguished on the external forwarding devices, which cannot complete the fine traffic management.

3) The vSwitch realizes high-performance network forwarding and monitoring. To get rid of the performance and flexibility limitations of packet switching in kernel space, vSwitch is gradually introduced. The vSwitch is a user-space program independent of the Linux kernel. It contains the complete I/O operations, switching, forwarding and monitoring logic, that gives the vSwitch great flexibility. Therefore, the packets in the virtual network can be efficiently classified, forwarded and monitored in the vSwitch, and do not worry about the limitation from kernel version.

Benefiting from the idea of SDN, software vSwitch enhances the ability to manage complex virtual networks on the cloud[54]. In terms of implementation, SDN greatly simplifies the table types that vSwitch needs to store and query. Traditional routing tables and Access Control Lists (ACLs) are replaced by unified flow tables. In the abstraction of the flow table, the vSwitch follows the processing logic of "match+actions". During the packet matching, vSwitch does not need to pay attention to the specific operation performed for the packets. After the matching process is completed, vSwitch performs the corresponding actions for the packets according to the matched flow entry. The action operation includes rich functions such as forwarding, rate limiting, and security configuration, which makes the CSP flexible to manage network configurations. At the same time, since the vSwitch is a third-party component independent of the Linux kernel and the hypervisor virtualization platform, it is also more suitable for business deployment, configuration and traffic monitoring at different levels such as per port, per flow and per packet.

In terms of performance, vSwitch also widely adopts user-space accelerated drivers to provide high-performance virtual networks. To reduce the packet copying times from VM to the physical NIC, existing vSwitches have shifted from traditional kernel modules to a pure user-space component to leverage kernel bypass technology[55, 56]. On the one hand, it can reduce the number of memory copying times during the packet delivery procedure, and on the other hand, it is more convenient to use a series of user-space components, such as DPDK[11], netmap[57] and Vector Packet Processing (VPP)[58], to accelerate packet processing. For example, the most widely used Open vSwitch[59, 60], VALE[61], BESS[62], snabbswitch[63], and Microsoft Cloud's VFP[64] all adopt the design of pure user-space data path.

Nowadays, the typical user-space vSwitch architecture and logic are shown in Fig. 2.2. Each virtual or physical NIC is connected to the vSwitch in the form of a port, and the packet processing logic driven by these devices is also integrated into the vSwitch. In vSwitch, the most important module is the Polling Mode Driver (PMD) thread that executes the main workload. It runs in a polling mode, traversing each port to receive packets, process and forward them. The running logic of the PMD thread is shown in the shadow part of Fig. 2.2.

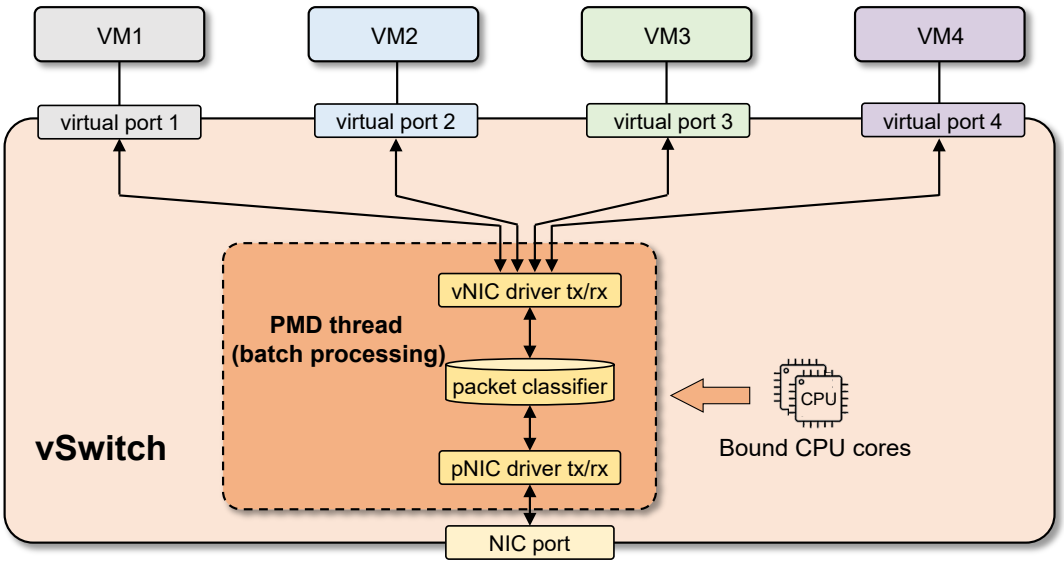


Figure 2.2 The typical vSwitch structure and logic

It first collects a batch of packets on each port, then looks up in the flow table to query the destination port, and finally calls the driver on the destination port to send packets out. Since the physical resources that can be used in vSwitch for network forwarding are limited, the core working modules (see the orange part in Fig. 2.2) are all shared to improve efficiency.

Through the above description and analysis, it can be seen that in order to realize the network virtualization for VMs, CSPs have gone through the stages of using TAP/TUN virtual device communication, Linux Bridge switching, and finally developed today's powerful user-space vSwitch. The widespread deployment of vSwitches also marks the maturity of virtual switching technology in large-scale cloud networks, and the problems of performance and flexibility that have plagued CSPs for a long time have been resolved. However, it also needs to be recognized that the current vSwitch has just solved the basic problem of whether it can provide tenants with virtual network services. When the resource integrated with vSwitches is getting higher and a series of sharing designs are adopted, how to provide a secure and stable virtual network environment for tenants has become a new challenge for CSPs.

2.1.3 The isolation requirements

As the goal of cloud computing is to allow lots of tenants to deploy VMs flexibly and elastically on the same platform, isolation and interference avoidance are the most basic requirements. At

the virtual network level, the requirements for isolation are reflected in three aspects: network performance, network failure, and security.

The network performance isolation in the vSwitch is mainly reflected in the network QoS guarantee for tenants. Usually, when a tenant purchases a VM on the cloud platform, an SLA network performance is included, that is, the network bandwidth and latency indicators. The vSwitch needs to ensure that the tenant can always achieve the purchased network performance when using the VM. On the other hand, SLA network performance is also an important basis for differentiated services. Therefore, the network performance isolation requires that the vSwitch can always ensure the tenant's stable SLA network performance when forwarding traffic for high-density deployed VMs.

Network failure isolation in vSwitch refers to the prevention of data plane DoS attacks that paralyze innocent tenants' network communication due to tenant's actions. In the forwarding process of the vSwitch, the TX/RX operations are relatively simple and implemented by the device drivers, so there is no resource sharing and possible common failure points. However, the packet classification module is much more complicated, and it contains many shared data structures to ensure high efficiency, which leads to the possible common failure points. If the packet classification process in the shared flow table fails, the normal packet forwarding of all tenants will come down. Therefore, for the flow table structure design in the vSwitch, it is necessary to provide tenant-level isolation, so as to prevent the common failure point from affecting innocent VMs on the same platform.

The security isolation in the vSwitch is mainly reflected in the defense against possible attacks in I/O operations and memory access. In the virtualized environment, security has always been the most concerned issue for CSPs, such as I/O edge channel attacks and illegal memory access. Most of these attacks are based on memory access, so there is a potential attack surface in vSwitch for that it directly takes over large amounts of device memory and driver logic. Since the memory operations in vSwitch mainly occur in packets I/O, the I/O operation design needs to ensure the highest security isolation as much as possible, and cannot be exploited to create attacks.

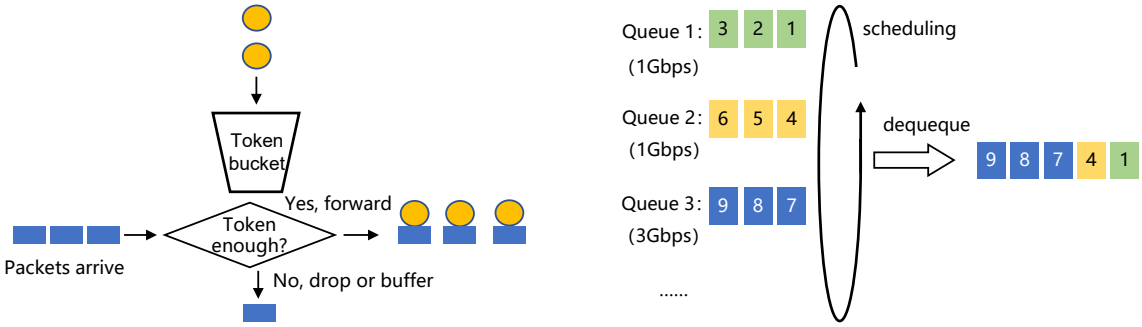
The rest of this chapter presents the existing works on these three isolation guarantees, as well as their remaining deficiencies.

2.2 Works on network performance isolation

Performance isolation is a relatively old topic and has been researched in hardware switches for a long time. This section will introduce the QoS method and its theoretical basis used on hardware switches to ensure tenant network performance isolation, and then analyze the difficulties and deficiencies of these existing methods applied to vSwitches.

2.2.1 Network QoS methods

As an important means of realizing tenant network performance isolation, the network QoS guarantee method has undergone decades of research, and a mature theoretical system has been established to support differentiated network services. From the implementation principle, it can be divided into two types of mechanisms: token bucket mechanism and fair queue mechanism.



(a) Token bucket (b) Fair queue

Figure 2.3 The principles of two kinds of QoS mechanisms

The token bucket mechanism is a lightweight method to ensure tenant network bandwidth. Its main idea is to accurately limit the sending rate according to the definition of bandwidth ($bps = passed\_bits/seconds$). The examples include the qdisc[12–14] token bucket mechanism in the Linux kernel, and single/two rate three color marker (srTCM/trTCM)[65, 66] token bucket algorithm in RFC standard. These token bucket mechanisms may have some differences in implementation, but the basic principles are generally the same. As shown in Fig. 2.3(a), a token bucket structure is allocated to each tenant in the forwarding device, and the number of tokens (such as bits) is stored in the token bucket. The token generation rate is determined by the bandwidth purchased by the tenant. Taking a tenant with 1 Mbps bandwidth as an example, the token bucket can generate 1M bits of tokens per second. When the tenant has packets to send, the forwarding device will check the number of tenant's tokens. The number

of tokens left in the bucket is used as the basis to decide how many bits of packets can be forwarded. The packets that exceed the number of tokens will be selectively discarded or left in the queue to wait for the next forwarding opportunity according to the policy. In the token bucket mechanism, most of the existing works focus on how to determine the bucket depth to face the bursty traffic[67], how to achieve robustness[15], and how to avoid conflicts, etc. But no matter how to improve, token buckets can only limit bandwidth, and cannot guarantee the packet latency along with fairness.

Compared with the token bucket, the fair queue mechanism is much more complex and involves more diverse scheduling algorithms. Its core function is to queue packets of different tenants according to the priority, weight and other factors, and then send packets out in order. Since the total bandwidth of the forwarding device is fixed, the fair queue can not only limit the sending rate of each queue, but also ensure the latency and fairness of the packets in different queues. As shown in Fig. 2.3(b), when there are three queues of packets to be sent, the forwarding device takes different numbers of packets from each queue in turn, and thus forms a queue with specific order on the sending port. So it not only guarantees bandwidth but also avoids blocking and unfair waiting. The granularity of the fair queue's early implementation is relatively coarse, and it is based on packets, which cannot cope with variable-length packet traffic. Therefore, a series of fine-grained scheduling mechanisms were proposed, such as GPS[16], DRR[17] and WFQ[18], etc. Currently, the existing works dedicate to the efficient implementation of these mechanisms [68, 19, 20].

2.2.2 Limitations

However, these well-studied QoS guarantee methods are all based on the premise that the performance of the forwarding device is constant, that is, the total bandwidth is constant. It is generally applicable on traditional hardware switches, but in software vSwitches using CPU cores to forward packets, this premise no longer exists.

In the vSwitch, the forwarding capacity of the CPU core processing engine is variable, and the CPU overhead required for forwarding different packets often varies by more than ten times[23]. Therefore, the packet forwarding tasks of different tenants will compete for the variable processing capacity in vSwitch. Even worse, only limited CPU cores can be bound to PMD threads in vSwitches for packet forwarding, because CSPs need to sell most of the CPU resources to tenants for maximizing profits. For example, in the deployment case of Google Cloud, only two CPU cores are used to allocate to PMD threads for forwarding[69], which further aggravates the competition for resources. The latest work of Cisco and Google Cloud

both revealed that the traditional QoS method cannot cope with the competition of physical resources and volatile forwarding capability in vSwitches. But these works only add a module of CPU resource isolation in addition to the traditional QoS module. Unfortunately, this newly added module still cannot solve the uncertainty caused by the variable forwarding capacity in vSwitch, so it is difficult to meet the service stability requirements in cloud networks.

2.3 Works on network failure isolation

Tenant network failure isolation is one of the most important capabilities in cloud networks, and one of the most common network failures is caused by DoS attacks that paralyze forwarding devices. However, since the introduction of SDN, the decoupling and interaction design of its data plane and control plane has provided new possibilities for DoS attacks. This section will introduce in detail the DoS attacks and defense mechanisms under SDN, and then focus on their real implementations in vSwitch.

2.3.1 Data plane DoS attack and defense

The SDN has been popular because it gives the network flexible programmability since it was proposed ten years ago. The basis of its flexibility lies in the decoupling of the data plane and control plane. The CSPs only need to configure and manage the rules on the control plane, and they can realize the management of each forwarding device in the complex network. One of the most important features is that whenever the forwarding device receives a packet that fails to match in the local flow table, the device will send a "packet_in" message to the controller. Then the controller configures rule and delivers the corresponding flow table entry to the local flow table of the forwarding device. This feature simplifies the network configuration and ensures network self-learning, but it also brings a new attack surface to the data plane.

Some works have warned that the interaction mechanism of the data plane and control plane in SDN will bring new security threats[24, 25]. The threat model is shown in Fig. 2.4. A malicious tenant only needs three steps to achieve a DoS attack on a switch and even the entire data plane: 1) malicious tenant sends carefully constructed traffic consisting of packets with the quintuples distributed very discrete; 2) When the traffic reaches the forwarding device for flow table lookup and forwarding, the packets will frequently miss in the local flow table and thus “pull” a large number of flow table entries from the controller; 3) Since new flow table entries are frequently added to the forwarding devices, there may be a shortage of storage, or the

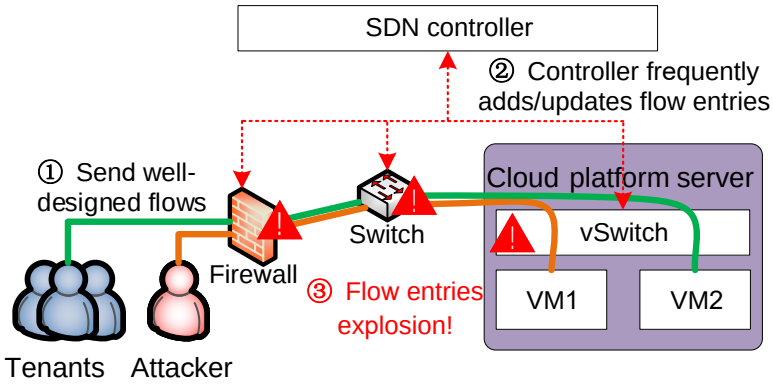


Figure 2.4 The principles of data plane DoS attacks under SDN

complexity of packet classification algorithm may increase. For example, in a hardware switch, frequent swap-in and swap-out operations may occur because the TCAM memory is full, which affects the packet matching operation of other normal traffic; while in a software-implemented forwarding device, the time complexity of packet classification algorithm is changed due to adding too much rule entries which results in a substantial decrease in the overall matching performance. Although there are little differences in attacks on the software and hardware devices, the results are the same–the forwarding devices suffer from DoS attacks, thus affecting the network experience of all tenants.

To defend the DoS attacks leveraging SDN rule update mechanism to cause failures in the flow table of forwarding devices, existing works attempt to detect and block the attack flows[26, 27]. Due to the attack traffic rate required for frequently "pulling" rules into flow table to form the attack is not low, some works try to use big flow detection mechanisms (such as sketch, etc.) to detect and prevent the malicious traffic. Another idea is to limit the frequency of the controller issuing rules to the local flow table, so that the malicious traffic cannot achieve the purpose of "pulling" a large number of useless rules into the flow table in a short time.

2.3.2 Limitations

However, most of these attack methods under SDN are too idealized and not close to the real environment, so they have received little attention. Moreover, the mainstream hardware switches can also be set to allocate independent flow table storage space for different flows, ports, and tenants. The flow table update behaviors of different tenants do not interfere with each other, so the DoS attack on these devices can hardly work. But inside the software vSwitch, things are much different. For efficiency, the resources and structures like flow tables are shared

to achieve higher performance, so compared with hardware switches, the vSwitch will face more possible DoS attacks.

Recently, Csikor et al. in [28, 29] implemented a Tuple Space Explosion (TSE) attack against vSwitches. Malicious tenants only need to construct packets with random headers and send low rate traffic, the DoS attacks can be carried out on the widely used OVS-DPDK. Under the attack, OVS-DPDK will pull down a large number of useless flow table entries from the controller, resulting in a sharp increase in the time complexity of the Tuple Space Search (TSS)[70] classification algorithm. Experiments show that the TSE attack can cause the forwarding failure of the vSwitch and reduce the network performance of all tenants by 80%.

The existing defense mechanisms cannot isolate common failure points and prevent this kind of TSE attack. Since TSE attack only requires a low attack traffic rate, the defense mechanisms such as big flow detection can be easily bypassed[26]. Other solutions will greatly affect the QoS of the vSwitch while preventing the attack. For example, in the case of limiting the frequency of “pulling” rules, the vSwitch may not be able to withstand traffic bursts, and even cannot serve concurrent access to the website well.

2.4 Works on memory security isolation

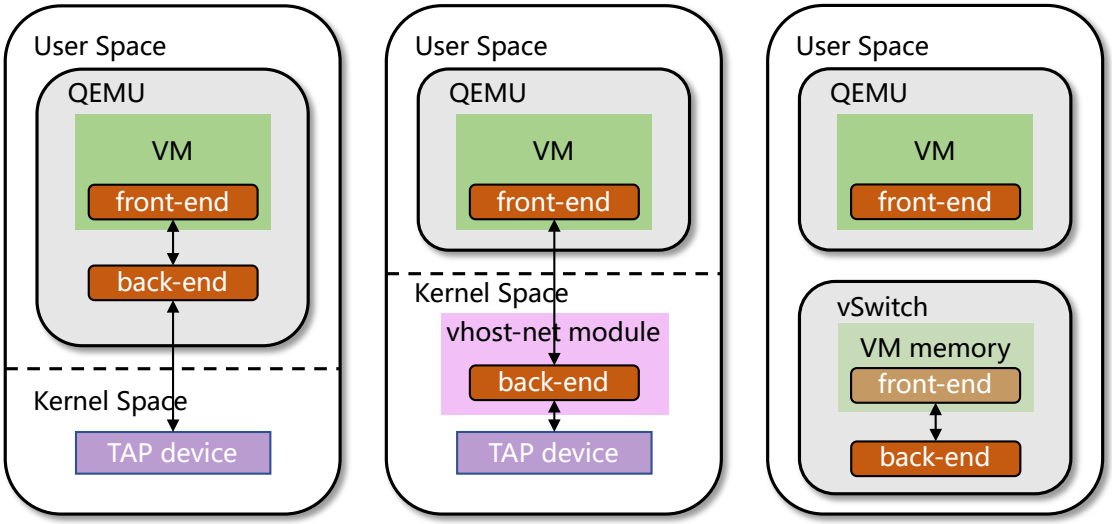
Secure isolation problems mostly occur in I/O operations and memory access behaviors. This section will introduce the evolution of VNIO and the memory isolation issues it introduces. Then, an analysis of related solutions and their limitations will be given.

2.4.1 Isolation issue in virtualized network I/O solutions

VNIO is an important means of transmitting packets between the VM and the host. A packet sent from the VM should be transferred from the VM memory to the host side memory through the VNIO technology. After that, the subsequent search and forwarding operations can be performed in vSwitch. In the past decade of exploration and evolution, the expectation for this technology lies in how to greatly improve performance.

Initially, the VNIO technology was realized by the hypervisor. This solution uses hypervisor processes such as QEMU to simulate interrupts for realizing the packet transmission. But the "VM-Exit" operation and multiple memory copying times result in poor performance. So in 2005, IBM proposed the para-virtualization standard–virtio[71] to solve the VNIO performance

dilemma. After three generations of improvements, the virtio para-virtualization standard has become a prerequisite for deploying VMs on the cloud platforms.



(a) virtio-net (b) vhost-net (c) vhost-user

Figure 2.5 The evolution of virtio-based para-virtualized network I/O mechanisms

Fig. 2.5 shows the evolution of virtio based VNIO. In the virtio standard, the function of VNIO is completed by the "front-end" and "back-end". The front-end is driven by the vNIC inside the VM and will not be easily changed, while the back-end is the actual performer of packet copying and I/O operations. With the continuous pursuit of performance, the implementation and position of the back-end are constantly changing. In the first-generation solution (see virtio-net in Fig. 2.5(a)), the back-end is integrated into the QEMU process. In this solution, each QEMU process performs the packet copying tasks for one VM inside it, and then writes the packet to the TAP device for virtual switching in Linux Bridge. However, this solution brings too much burden to QEMU, and even the normal operation of the VM is affected in the case of high-speed network communication, so the researchers proposed the second-generation vhost-net mechanism[72]. As shown in Fig. 2.5(b), in the vhost-net mechanism, the back-end is transferred from QEMU to the vhost-net module in the kernel space, and this kernel module will uniformly respond to all VMs' network I/O requests. With the rise of kernel bypass and high-performance user-space vSwitches in recent years, the vhost-net mechanism has gradually moved up to the user space and evolved into vhost-user (as shown in Fig. 2.5(c)). Under the vhost-user mechanism, the vSwitch realizes the back-end function through mapping all the VMs' memory to its own memory space. The whole packet transmission is completed in the user space without interruption and privileged instructions, thus greatly improving the I/O performance and becoming the mainstream deployed mechanism in the industry.

Throughout the evolution of virtio-based VNIO mechanisms, it can be seen that too many components and even memory isolation problems are gradually introduced due to the pursuit of ultimate performance. In both virtio-net and vhost-net solutions, the privileged processes are used to implement back-end I/O functions. For example, virtio-net chooses QEMU, which is originally the hypervisor process responsible for managing VM memory, to implement the back-end function; while in the vhost-net, the back-end function is implemented by a kernel module, which also has sufficient permissions to access the VM memory. Thus, there is no memory isolation issue in the previous two generations of mechanisms. However, in the vhost-user, the vSwitch process is a user-space process and is restricted from accessing VM memory. In order to implement I/O operations, the memory sharing mechanism (the VM memory is completely shared with the vSwitch) is introduced. That brings the risk of illegal memory access and attacks to the VMs while providing a high-performance data path.

2.4.2 The limitations of reinforcement works

There have been studies and warnings on the risk of illegal memory access in the vhost-user mechanism. From the open-source community, Wang et al. reported a DMA attack based on the shared memory of vhost-user[39, 41, 40]. As long as malicious tenants take advantage of vulnerabilities or software flaws to control the user-space vSwitch, they can access all other tenants' memory without restriction. To solve this issue, a solution called vIOMMU is proposed. Its idea is to create a vIOMMU virtual device in the QEMU process to implement the address translation and legality check for vSwitch memory access. When the vSwitch needs to access VM memory, it first needs to send a request to QEMU for legality check and address translation, so as to prevent illegal memory access to the VMs[41–43]. However, this vIOMMU solution will greatly increase the inter-process communication overhead. Experimental results show that it will reduce the system performance to 20% of the original, which is unaffordable for CSPs.

At the same time, some other VNIO mechanisms have been proposed in academic papers, such as NetVM[34], IVSHMEM[35, 36] and ClickOS[73]. But these mechanisms all use shared memory to speed up packet copying, which also causes isolation risks and security problems. In addition, due to the lack of standardization and community support, they also have shortcomings in compatibility. Therefore, none of these solutions can be used for cloud platforms

Chapter 3

C2QoS: CPU-cycle Isolation based Network QoS Method

3.1 Introduction

As the foundation of cloud computing, virtualization technology allows tenants to flexibly deploy various services on the cloud platform in the form of VMs. In the cloud virtualization environment, VM deployment density is usually high, that is, a large number of VMs belonging to different tenants and running different services are deployed on the same physical server. These VMs share various resources of the host, including CPU, memory, and network. Therefore, how to design mechanisms to allocate these resources and provide differentiated services is the most concern for CSPs.

On the server of a cloud platform, VM network resources are provided by software vSwitch. The vSwitch connecting NIC to the VMs provides all VMs' network connectivity. All packets sent from the VMs need to go through steps such as classification and forwarding in vSwitch before they can be sent out via the NIC. Therefore, the upper limit of bandwidth and latency these VMs can achieve are determined by the processing capabilities of the vSwitch[64, 59]. But for a CSP, it is a common practice to increase the share of CPU resources with VMs, and thus very limited CPU resources are left for vSwitch's forwarding tasks, e.g., the Google cloud uses no more than two dedicated physical cores to perform forwarding tasks[69]. As a result, all the VMs compete with each other for the limited processing capacity of the vSwitch, essentially for the CPU resources occupied by the vSwitch.

Meanwhile, the CPU resources needed to maintain particular network performance are hard to predict in vSwitch. Different from the hardware switch that has a constant forwarding capacity, the vSwitch's processing capacity with these IO-dedicated CPU cores is variable when being used to forward traffic with different characteristics. For example, at the same bps rate, compared to forwarding the traffic with 1518-byte packets, forwarding the traffic with 64-byte packets consumes 10 times more CPU cycles[22, 23]. As a result, the vSwitch can hardly guarantee all tenants' network SLA performance in all situations.

Existing network QoS strategies in software vSwitch are inherited from the interface-based solutions of hardware switches, and do not consider the issues of CPU resources competition among tenants, as well as the variable vSwitch forwarding capacity. As a result, they cannot ensure VMs' SLA performance targets. Our experimental results on a multi-tenant cloud platform in Section 3.2, show that the innocent VM bandwidth can be decreased by up to 20% due to the competition of IO-dedicated CPU usage. In terms of latency, all VM's latency increase hundreds of times, since all VMs' forwarding tasks compete for the limited CPU cores in terms of timing and are not processed in a differentiated manner. Some works[21, 22] have noticed the resources competition issue, and they added a module for CPU resources isolation before the QoS module. The effects of these works were limited because they are still interface-based and the variable vSwitch forwarding capacity is still ignored.

Different from existing solutions, in this chapter we propose a new CPU-Cycle based QoS strategy (C2QoS) to completely solve this issue. As the "virtual" network resources are realized by the IO-dedicated CPU cores, we guarantee VM's network SLA performance by directly apportioning these CPU cycles to VMs. The challenges of achieving this goal include: 1) How to establish a correspondence between VM's bandwidth and CPU usage. 2) How to assign CPU cycles to VM to strictly guarantee its bandwidth. 3) How to ensure the SLA latency, especially for the delay-sensitive applications (e.g., the web and video servers require lower response latency than file system servers).

To address these challenges, this chapter makes the following main contributions:

- We propose a modeling methodology to build the correspondence between forwarding capacity and CPU resources in vSwitch. The model characterizes the effect of different working conditions over the vSwitch forwarding capacity. These conditions include the tenant traffic characteristics, as well as the deployment configurations.
- Based on the model, we propose the C2QoS strategy, containing a CPU-Cycle based Token Bucket (C2TB) mechanism for performing isolation enhanced rate limiting and a Hierarchical Batch Scheduling (HBS) mechanism for providing latency guarantee.

- We implement the C2QoS strategy on the OVS-DPDK[11, 59] platform. The experiments on a multi-tenant cloud platform show that compared with existing strategies, the influence of CPU resource congestion on bandwidth is eliminated and that on latency is reduced by 80%.

3.2 Problem define and motivation

3.2.1 Network QoS in vSwitch

Network QoS strategy is a well-studied topic in hardware switch, and a lot of works have been proposed. According to the implementation and function, the QoS strategies can be divided into two types: the token bucket mechanisms, and the fair queuing mechanisms. The token bucket mechanisms are used to limit the bandwidth sharing with little overhead, but they cannot ensure latency[12–14, 67]. In contrast, the fair queuing and traffic scheduling mechanisms represented by GPS, WFQ, and DRR, are proposed to guarantee SLA bandwidth and latency more finely, while bringing relatively high complexity[16, 18, 17].

When realizing these two kinds of QoS strategies in the hardware switch, the sufficient processing capacity inside the switch brings significant advantages. The main reasons are argued in [15]: the overhead of processing each packet is fixed; the token buckets and queues are implemented by hardware and they can complete the corresponding functions without compromising the performance; high-precision clock and hardware feedback support[74, 75].

Unfortunately, none of the above advantages exists in software vSwitch. As mentioned in Section 3.1, the CPU cores left for vSwitch are limited, and meanwhile their processing capacity is variable when being used to forward traffic with different characteristics. For example, in Google's experiments, forwarding a flow with 64-byte packets at a speed of 512 Mbps will consume more CPU cycles than forwarding a flow with 1518-byte packets at a speed of 2.4 Gbps[22]. On the other hand, as a software-based process, the vSwitch has particular bottleneck and resource competition points, which are completely different from hardware switches. These differences make that the QoS strategies inherited from hardware switches cannot work well in the vSwitch. We will further demonstrate in this section that, existing QoS solutions can cause performance issues in both bandwidth and latency.

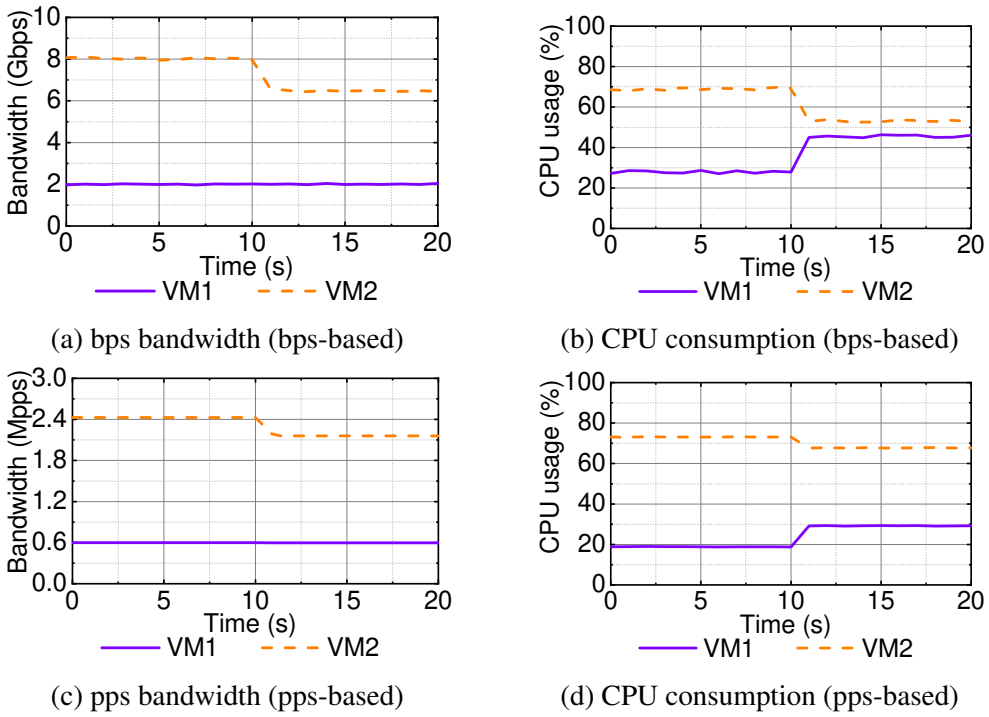
3.2.2 Bandwidth issue

The existing rate limiting methods in the vSwitch of cloud servers usually use the light-loaded token bucket for efficiency, e.g. the MBFQ rate limiting method used by Microsoft is implemented with a token bucket algorithm[15]. These existing token bucket mechanisms are all based on bps or pps, and directly limit the number or bits of packets that can be forwarded to guarantee tenants' SLA bandwidth. As the IO-dedicated CPU resources are limited and the processing capacity is variable when being used to forward different traffic, one tenant may legally squeeze the CPU resources and harm the bandwidth of others.

In this section, we adopt the best-performing three color marker (TCM) rate limiting algorithm[67, 65, 66] in the OVS-DPDK platform to demonstrate the issue. On one server, we launch two VMs to connect to the OVS-DPDK as the sender, and then use another directly connected server with the same hardware configurations as the receiver. We use pkt-gen from netmap[57] inside the two VMs as packet generators. It should be noted that all experiments in this chapter use the same platform configurations: Intel Xeon CPU E5-4603 v2 2.20GHz (32 logical cores on 4 NUMA nodes), 64GB DDR3 memory at 1333MHz, one Intel 82599ES 10-Gigabit Dual Port NICs and Ubuntu 16.04.1 (kernel 4.8.0) as operation system. The cloud platform is built on QEMU 2.10, DPDK 17.11.2 and OVS 2.9.2. Every VM is assigned with 2 GB memory and 1 logical CPU core.

In Fig. 3.1(a) and Fig. 3.1(b), we show how the bps-based token bucket mechanism fails to ensure VM bandwidth. The VM1 and VM2 share one dedicated CPU core in OVS-DPDK for forwarding, and their bandwidths are limited to 2 Gbps and 8 Gbps respectively. Within the first 10 seconds, they send 512-byte packets and their bps bandwidths are precisely limited. Starting from the 10th second, VM1 sends small packets (changing the packet size to 64-byte). In order to achieve the same bps throughput (2 Gbps) as before, VM1's forwarding tasks in vSwitch consume 20% more CPU resources as shown in Fig. 3.1(b). This leads to a drop in the CPU consumption of VM2's forwarding tasks, which in turn reduces VM2's available bandwidth. Eventually, the innocent VM2 was affected by the tenant behavior inside VM1, resulting in an approximately 20% decrease in VM2's bandwidth.

The same situation also occurs in the pps-based token bucket mechanism. In Fig. 3.1(c) and Fig. 3.1(d), VM1 and VM2 also share one CPU core in OVS-DPDK for forwarding, and their bandwidths are limited to 0.6 Mpps and 2.4 Mpps respectively. Within the first 10 seconds, they behave well and both send a single flow. The VM1 starts to send multiple flows from the 10th second (change configurations in pkt-gen), which makes it require more CPU resources as the packet classification in OVS-DPDK gets slower. Similarly, to achieve the previous pps



(a) bps bandwidth (bps-based)

(b) CPU consumption (bps-based)

(c) pps bandwidth (pps-based)

(d) CPU consumption (pps-based)

Figure 3.1 The bandwidth isolation issue in existing token bucket mechanisms

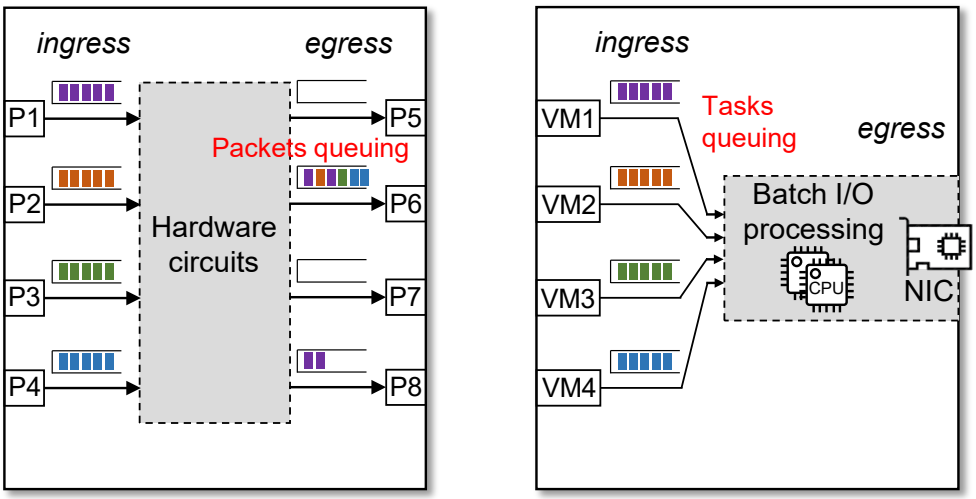
bandwidth, VM1 preempts part of the CPU resources belonging to VM2 as shown in Fig. 3.1(d), which leads to a 16% decrease in VM2 bandwidth.

These two experiments demonstrate that, even the tenant behavior inside a small-weight VM can influence the other innocent VMs' bandwidth. The reason is that the VM forwarding tasks compete for the IO-dedicated CPU cores in vSwitch, which is ignored in the existing rate limiting mechanisms. By exploiting this flaw, greedy tenants can obtain more CPU resources, or an attacker can construct specific traffic to harm the network performance of all tenants on the server. In either case, CSPs cannot provide the well-behaved tenants a stable network performance.

3.2.3 Latency issue

In addition to the bandwidth issue, the existing traffic scheduling mechanisms inherited from the hardware switch also cause latency issues in vSwitch. As processing engine and logic are very different between the hardware switch and the vSwitch, the resource competition occurs at different stages. Simply applying the previous strategy cannot solve the competition problem on a different platform.

As shown in Fig. 3.2, we present the abstract of packet processing logic in the hardware switch and the software vSwitch to analyze the different requirements of scheduling in the two architectures. In the hardware switch, the circuits with powerful processing capabilities make the resource contentions mainly occur at the *egress* stage, where the traffic of multiple in_ports is gathered for sending out on a particular port (see port P6 in Fig. 3.2(a)). The existing traffic scheduling mechanisms working at this stage[20, 68, 76] can queue up packets of different in_ports so as to guarantee differentiated latencies. However, in the software vSwitch, the packet processing capacity of CPU cores is far inferior to the hardware circuits that can achieve line speed. With the common practice in the cloud platform that very limited CPU cores are used for vSwitch, concurrent VMs compete for these CPU cores to execute the expensive batch I/O processing in the *ingress* stage (see Fig. 3.2(b)). Due to the absence of task scheduling at the *ingress* stage, VMs indiscriminately queue up for batch I/O tasks to be completed, which causes mutual influence and high latency. In the worst case, on a server running n VMs, each VM will suffer from the additional latency caused by $n-1$ times of batch I/O processing.



(a) hardware switch (b) software vSwitch

Figure 3.2 The comparison between hardware switch and software vSwitch

We also use experiments to demonstrate this issue. To simulate the multi-tenant scenario on the cloud, we increase the number of VMs to 16 on one server and measure the TCP latency. One CPU core is used to forward traffic in the OVS-DPDK. To measure TCP latency, we run qperf[77] as a client-side program in all VMs simultaneously, while the server-side program is run in another directly connected physical server. We measure 20 sets of data for each experiment to avoid accidents.

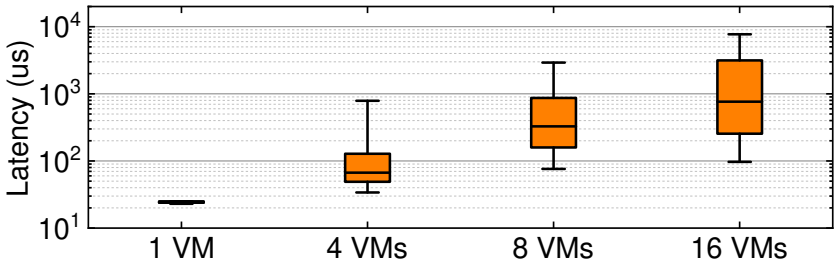


Figure 3.3 The TCP latency with different numbers of VMs deployed

The results are shown in Fig. 3.3, where the latency in each case is shown in the form of a box diagram. It can be seen that when there is only 1 VM running on the server, its TCP latency is stable and maintained at 26–27 us. But with the number of VMs growing, the TCP latencies become unstable and all increase exponentially. When deploying up to 16 VMs on this server, all VMs suffer from hundreds of times higher latency indiscriminately due to waiting for the one IO-dedicated CPU core to sequentially process other VMs' batch I/O processing at the *ingress* stage.

3.2.4 Motivation

The reason that VM's bandwidth and latency cannot be guaranteed is that the existing QoS strategies ignore the IO-dedicated CPU resources competition inside the vSwitch. The lack of management and apportionment of CPU resources brings a series of flaws including bandwidth isolation and undifferentiated high latency. Some previous works have mentioned this issue, and some solutions have been proposed, e.g., Addanki et al. [21] considered separately apportioning IO-dedicated CPU resources and bandwidth on the software router, and Kumar et al. [22] proposed a method by using a CPU-based weighted fair queue to isolate CPU competition among VMs. But all of these works have limited effects because they only add a CPU isolation module before or after the existing interface-based QoS mechanisms, but fail to consider the variable vSwitch forwarding capacity and the different resource competition points in the software vSwitch process.

Essentially, the network forwarding capacity of vSwitches is not a kind of physical resource, but a kind of "virtual" resource provided by IO-dedicated CPU resources in the vSwitch. Starting from this point, the motivation of this work is to adopt the CPU resources apportionment, that reflects the network forwarding capacity more directly, in the VM network QoS solution. In order to do that, we first propose a modeling methodology to build the relationship between

CPU resources and network forwarding capacity in vSwitch. Based on the vSwitch network performance model, we design and implement a new VM network QoS strategy.

3.3 Bandwidth–CPU model

To guide the design of QoS strategy, we first need to model the correspondence between forwarding capacity and CPU utilization in vSwitch. Our modeling of the vSwitch forwarding procedure is based on the OVS-DPDK platform, which is a state-of-the-art implementation and has been widely adopted by the industry. As the OVS-DPDK platform represents a lot of vSwitches in terms of packet processing logic, the modeling method can be easily applied to other vSwitch platforms.

3.3.1 Packet forwarding procedure in vSwitch

In OVS-DPDK, several PMD threads are launched and bound to the limited IO-dedicated CPU cores. For efficiency, these PMD threads use batch processing mode to process tasks.

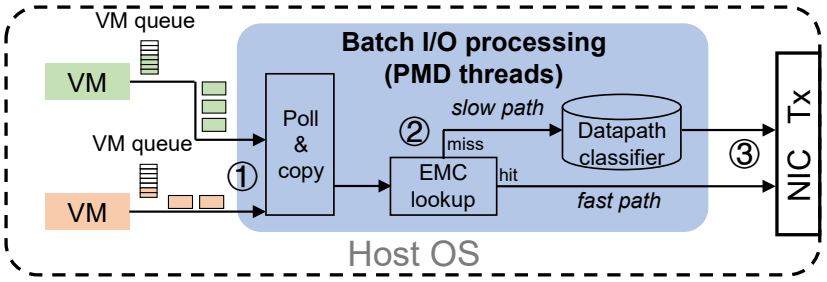


Figure 3.4 The processing logic in OVS-DPDK

As shown in Fig. 3.4, the batch I/O processing procedure in the OVS-DPDK consists of three stages delivering packets from the VM to the external network. The first stage is *ingress*, the PMD thread copies a batch of packets from the VM memory to the vSwitch's buffer. Next, in the *classification* stage, the PMD thread looks up their destination port based on the five tuples. If the five-tuple is found in the Exact Match Cache (EMC), we go to the next stage. But if it is missed, the PMD thread will use more CPU cycles to look up in the more comprehensive classifiers (the datapath classifier in Fig. 3.4) and then go to the next stage. Finally, it is in the *egress* stage that the PMD thread writes the packet descriptors to the NIC queue, and then

the NIC can send packets out. According to these three stages, we also divide the CPU cycles consumed by the VM forwarding tasks into three parts as shown in the equation below:

$$C = C_{ingress} + C_{classification} + C_{egress} \quad (1)$$

Where $C_{ingress}$ indicates the CPU cycles consumed in the *ingress* stage, $C_{classification}$ refers to that consumed in the *classification* stage and C_{egress} corresponds to the CPU cycles consumed in the *egress* stage.

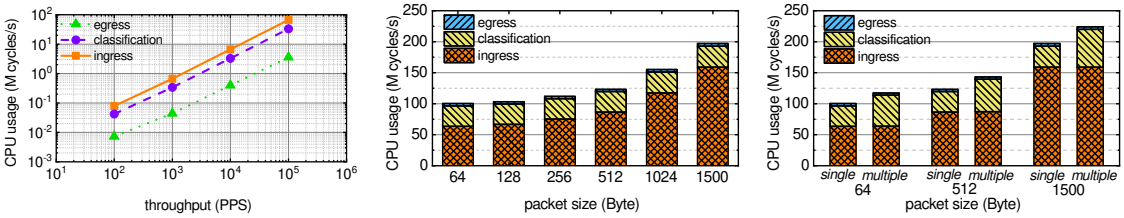
It is worth noting that in the *classification* stage, the EMC capacity is limited, e.g., it has only 8192 entries in OVS-DPDK, so it can only store the most recently searched five-tuples. The datapath classifier is the main body of the classifier algorithm like tuple-search-space (TSS)[70] and contains all the rules in the vSwitch. Each time a five-tuple search is hit in EMC, the $C_{classification}$ only contains lookup cost in EMC. But if a lookup is missed in EMC and hit in the datapath classifier, the hit entry needs to be added to EMC[59]. Therefore, the $C_{classification}$ under this case contains the lookup cost in EMC and datapath classifier, and the update cost in EMC. Obviously, the latter is much larger than the former, and the specific value will depend on the number and complexity of the rules.

3.3.2 Factors affecting CPU consumption

During the whole packet forwarding procedure in vSwitch, many factors can affect the CPU consumption in the three stages, e.g. throughput, the number of flows, the number of VMs, and so on. According to the main bodies that control them, we divide these factors into two types: traffic characteristics managed by tenants and VM deployment configurations managed by CSPs. We will experiment to study how the three parts of CPU consumption in Eq. (1) are affected by these factors, and then use the measurement-based methodology to model CPU consumption under different situations.

1) Impact of network traffic characteristics

The first factor we consider is the network traffic characteristics, which can be changed by the tenant behavior inside a VM: sending rate, packet size and the number of flows. We launch one VM on the OVS-DPDK platform and assign one CPU core as the IO-dedicated CPU resources on one server. The impacts of the three traffic characteristics on CPU usage for forwarding are shown in Fig. 3.5. During each experiment, we vary one characteristic and record the results, while keeping the other two with a certain value.



(a) CPU usage - pps (b) CPU usage - packet size (c) CPU usage - multiple flows

Figure 3.5 The impact of traffic characteristics on the CPU consumption for forwarding

Sending rate (pps). We keep the packet size at 1500-byte and keep the number of flows at 1 during this experiment. In Fig. 3.5(a), we find the CPU cycles consumed in all three stages are proportional to the pps. So this is the basis and premise of all existing bps/pps-based rate limiting methods: with no other traffic characteristics changed, the CPU competition will not occur.

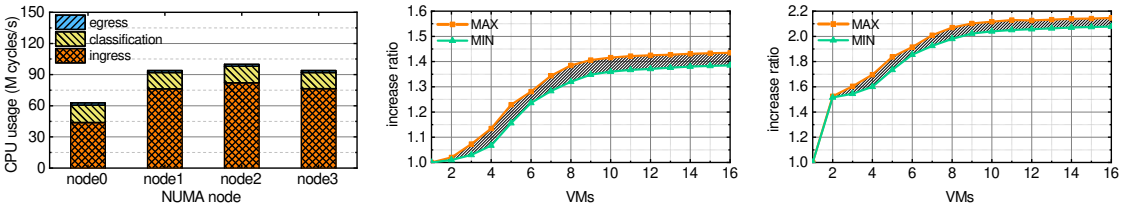
Packet size. In this experiment, we keep the pps at 10^5 and keep the number of flows at 1. The results of different packet sizes are shown in Fig. 3.5(b), and it can be seen that increasing the packet size will only increase $C_{ingress}$ and have nothing to do with $C_{classification}$ and C_{egress}. The increase of $C_{ingress}$ is due to the fact that only the stage *ingress* contains packet copying, so the larger packet requires more time to copy. For example, the $C_{ingress}$ under the case of forwarding 1500-byte packets is more than twice that of forwarding 64-byte packets at the same pps rate.

Number of flows. We keep the pps at 10^5 and keep the packet size at 1500 bytes during this experiment. The result of concurrent flows is shown in Fig. 3.5(c). Compared with only sending one flow ("single" in the figure), sending a large number of concurrent flows ("multiple" in the figure, we range dst ip from 0.0.0.0 to 255.255.255.255 and at the same time randomize the port number) will cause the packet classification frequently misses in EMC lookup and the packet will enter the longer search path, and thus $C_{classification}$ is increased. In our experiment, the $C_{classification}$ in the worst case is 1.67 times more than that in the best case. But it should be noted that, the number and complexity of the rules in the flow table will affect this ratio.

2) Impact of deployment issues

When deploying multiple VMs on the same physical server, some deployment configurations will influence the CPU consumption in packet forwarding. These deployment issues in the case of multi-tenancy scenario include: VM memory location (on which NUMA nodes[78, 79] of the physical server), the number of VMs on the same server and the number of IO-dedicated CPU cores. As these factors are mainly independent of each other, the influence can be expressed as

$\prod R_i * C_{single}$, where R_i represents the growth rate of CPU consumption under the influence of each factor.



(a) CPU usage - VM location (b) CPU usage - one core (c) CPU usage - two cores

Figure 3.6 The impact of real deployment issues on the CPU consumption for forwarding

VM memory location. The influence of NUMA architecture on memory access widely exists in today's commercial servers, so we need to evaluate it. We have described the CPU information in Section 3.2.2, and there are 4 NUMA nodes on our servers. In the experiment setting, the vSwitch's IO-dedicated CPU cores and memory are located on NUMA node 0. So the memory on nodes 1, 2 and 3 requires the CPU cores in vSwitch to access memory across nodes, which is slower than accessing the memory on node 0. At the same forwarding rate, the CPU cycles required by the forwarding task of VM deployed on each NUMA node are shown in Fig. 3.6(a). It can be seen the VMs on nodes 1, 2 and 3 need 40% more CPU cycles than the VM on node 0 to complete the forwarding task, and that is mainly due to the increase of $C_{ingress}$ by memory access across nodes. So for the VMs on nodes 1, 2 and 3, the coefficient R in this factor is 1.4.

Number of VMs. As the number of VMs grows, the competition on memory bus and cache will increase the CPU consumption of all VMs' three forwarding steps. As shown in Fig. 3.6(b), when the number of VMs is less than 8, the additional CPU consumption caused by competition for cache will greatly increase, while it will be almost the same after VM grows more than 10. The maximum and minimum curves in the figure show the CPU consumption increase ratio when deploying VMs on the same NUMA node (the worst case) and on different nodes (the best case). The former situation will lead to higher competition. The coefficient in this scenario is changeable and needs to be measured through actual experiments. For example, according to Fig. 3.6(b), if 4 VMs are deployed on the same NUMA node, the CPU resources consumed by each VM will be 1.14 times more than that under the single VM case. But if the 4 VMs are deployed on different NUMA nodes, the ratio changes to 1.07.

Number of CPU cores. Finally, we increase the number of IO-dedicated CPU cores to 2 in vSwitch and the result is shown in Fig. 3.6(c). Comparing Fig. 3.6(b) and Fig. 3.6(c), it can be found that using 2 logical cores for forwarding will consume about 1.47 times more CPU

cycles than one logical core for forwarding in any case. It is mainly due to competition for locks in the code, e.g. the synchronization among multiple PMD threads.

Therefore, when considering the CPU cycles assigned to a particular VM in the practical environment, CSPs need to multiply the C_{single}, the necessary CPU cycles measured under the single-VM case, by all the increase coefficients recorded in the above experiment results. For example, if 4 VMs are deployed in NUMA node 1, and two IO-dedicated CPU cores are assigned to forward traffic for them in vSwitch, the coefficients to be multiplied under the above three deployment configurations are 1.4, 1.14 and 1.47, respectively, according to Fig. 3.6. Only when all these factors are considered, the CPU cycles allocated to each VM can ensure its purchased bandwidth.

3.3.3 Modeling methodology

According to experiments and analysis, CSPs can build their bandwidth-CPU models in their vSwitch platforms. When the tenant requirements and configuration information are given, the CPU resources required for the tenant's VM to achieve SLA network performance can be calculated. In the following, we will present the modeling procedure and the required information to guide CSPs to implement in the real environment.

Firstly, the CSPs need to perform measurements in advance to establish a bandwidth-CPU model as described in Section 3.3.2. For the impact of traffic characteristics, some preset values can be selected for measurement and the results can be stored in a table. For example, the packet size can be {64, 128, 256, 512, 1024, 1500}, and the number of flows can be {single, multiple}. With this table, a particular input like (pps = 10000, packet size = 1024, number of flow = single) will get a certain output C_{single}. Next, for the deployment configurations, CSPs can also use the experiments to get the corresponding configuration and its coefficient R_i, and store it in tables. As the example shown in Section 3.3.2, given the input (the number of VMs = 4), we will get a coefficient R_i as 1.14.

After the measurement-based model is built, in the second step, the CSPs rely on two types of information to form the inputs of the model when deploying VMs: the traffic characteristic preference from the tenant's choice and VM deployment configurations. For the three traffic characteristics, they can be included in the SLA and tenants can choose them when purchasing the VM. As the iMIX traffic [80, 81] represents an average level of all tenants' traffic, CSPs can also set its characteristics as the default values to meet most tenants' requirements. For deployment configurations, CSPs can easily detect them. But as they may change frequently

with the creation and deletion of VM instances, that requires CSPs to change the inputs in real-time. Then according to the formula $\prod R_i * C_{single}$, the required CPU resources for each VM to achieve SLA network performance can be calculated based on the tables in the first step.

3.4 Design

Based on the bandwidth-CPU modeling methodology, we are able to design C2QoS strategy. The premise of C2QoS is that the number of VMs to be deployed is in accordance with the resources on physical servers and there is no overprovision. As each VM's required IO-dedicated CPU resources can be calculated based on the model, we define that under the C2QoS strategy, the deployment of VMs should follow two rules: the sum of all VMs' purchased bandwidth should not be more than the NIC bandwidth; and the sum of CPU resources that we calculate for each VM according to the model should not exceed the IO-dedicated CPU cores. Without these rules, the resource shortage will occur all the time, and no strategy can work.

In C2QoS, we propose the C2TB mechanism and the HBS mechanism to provide isolation enhanced rate limiting and hierarchical latency respectively. In this section, we will illustrate in detail the design.

3.4.1 CPU-cycle based token bucket mechanism

To guarantee VM bandwidth through the CPU resources apportionment, C2TB needs two steps: allocating the IO-dedicated CPU resources to particular VMs; using the allocated CPU resources to strictly limit the forwarding rate.

3.4.1.1 CPU resources allocation

Firstly, we construct a new kind of token bucket for each VM. Different from the traditional token bucket algorithms that use the bits or number of packets as tokens, the tokens in C2TB represent the remaining usable IO-dedicated CPU cycles of each VM. The token generation rate of each VM is the IO-dedicated CPU cycles/s allocated to it. We use C_{alloc} to indicate the CPU cycles/s required by each VM to achieve purchased bandwidth. Using the modeling methodology in Section 3.3, we can set the token generation rate to the fit value of C_{alloc}, and it can strictly ensure tenants' purchased bandwidth in practice.

Meanwhile, besides the C_{alloc}, the idle part of the IO-dedicated CPU resources also needs to be entirely allocated to VMs for the MIN-MAX bandwidth allocation policy[82, 15], which is widely used in industry. An example of the MIN-MAX bandwidth guarantee under C2TB is shown in Fig. 3.7. The MIN bandwidth (the purchased bandwidth) is ensured by only assigning basic C_{alloc} to the particular VM, while the MAX bandwidth is obtained by assigning the C_{alloc} plus C_{idle}, which means the idle CPU cycles of the IO-dedicated CPU cores. In the example, we assume the CPU resources required to achieve 1 Gbps and 2 Gbps bandwidth are 0.2G cycles/s and 0.4G cycles/s, respectively. After their purchased bandwidth are guaranteed, there are still 1G cycles/s left idle on the 2.2GHz CPU core and it can be fully assigned. For the 4 VMs, they can share the C_{idle} of 1G cycles/s according to their weights to achieve their MAX bandwidth. Therefore, the maximum CPU resources allocated to them are 0.36G cycles/s, 0.36G cycles/s, 0.72G cycles/s and 0.72G cycles/s, respectively.

C2TB	VM1	VM2	VM3	VM4
weight	1	1	2	2
purchased bandwidth	1 Gbps	1 Gbps	2 Gbps	2 Gbps
MIN(cycles/s)	0.2G	0.2G	0.4G	0.4G
MAX(cycles/s)	0.2G + 0.16G	0.2G + 0.16G	0.4G + 0.32G	0.4G + 0.32G

Figure 3.7 The CPU-cycle based token bucket mechanism

After we entirely allocate the IO-dedicated CPU resources to VMs, another problem may occur that the allocated CPU resources may overflow from the token bucket and be wasted, when VM's network load is light. In this condition, we also need to reallocate these unused CPU resources. For example, if one or two VMs in Fig. 3.7 are sleeping and no traffic is generated, their tokens will always overflow and this part of the overflowed tokens can be redistributed to other VMs according to their weights. We present this reallocation logic in the token update function as shown in Algorithm 1. It will be called at regular intervals (its value is *invl* in the function) to count the number of tokens in the token buckets for all VMs on this server. The variable *loop_unused_cc* is used to collect all the overflowed CPU cycles in this loop and at last it will be stored in *sum_unused_cc* for reallocation next time to update tokens. In the loop, each VM will calculate the number of tokens generated at the rate of $C_{alloc} + C_{idle}$ within the *invl* time. Then, each VM will get some overflowed CPU cycles based on its weight. We should point out that the $VM_i.weight$ in the algorithm is a ratio, calculated by dividing the VM's weight by the sum of all VMs' weights. So at last the tokens added into the token bucket

contain two parts: the generated tokens and overflowed tokens. The last step in the loop is to check whether the number of tokens exceeds the bucket depth. If yes, the overflowed part needs to be taken out and saved for next token update.

Algorithm 1 C2TB token update function

```
1:  function C2TB_UPDATE_CALLBACK
2:      loop_unused_cc ← 0
3:      unused_cc ← 0
4:      for i = 0 → VM_cnt do
5:          gene_cc ← invl ∗ (VM_i.C_alloc + VM_i.C_idle)
6:          unused_cc ← sum_unused_cc ∗ VM_i.weight
7:          sum_unused_cc− = unused_cc
8:          VM_i.tokens + = gene_cc + unused_cc
9:          if VM_i.tokens > bucket_depth then
10:             loop_unused_cc + = VM_i.tokens − bucket_depth
11:             VM_i.tokens ← bucket_depth
12:         end if
13:     end for
14:     sum_unused_cc ← loop_unused_cc
15: end function
```

3.4.1.2 Rate limiting

The last part of C2TB is to use the allocated CPU cycles in VMs' token buckets to limit their forwarding rates. In the traditional token bucket algorithm, the number or bits of packets are calculated during the batch I/O processing and the packets exceeding the available tokens will be dropped exactly. But in C2TB, since it does not know how many CPU cycles will be consumed, it is impossible to decide how many packets should be dropped. For efficiency, we adopt the following policy: we allow the number of tokens to be negative, and whether the tokens are greater than 0 determines whether this batch I/O processing task can be executed. For each VM, only if its tokens are greater than 0, it can send out a batch of packets, and the number of CPU cycles consumed is subtracted from its token bucket after the batch processing completes. As the CPU cycles used for each VM's packet forwarding tasks are reasonably assigned in C2TB, the VM bandwidth can be guaranteed with good isolation.

3.4.2 Hierarchical batch scheduling mechanism

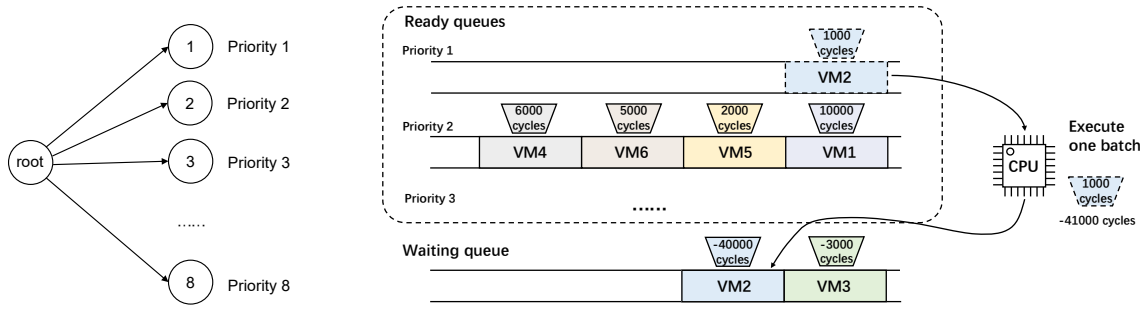
As the existing scheduling mechanisms only work at the *egress* stage and cannot avoid the high latency of CPU resources contention in the other stages, we turn to think about scheduling the entire batch I/O processing procedure (including *ingress*, *classification* and *egress*) for VMs. In the field of CPU task scheduling, we find the batch task scheduling model in vSwitch is much closer to the works in [83, 19]. These works schedule tasks on CPU cores to ensure that tasks with light load will not be blocked too long by heavy load tasks. Though these works still fail to make it flexible to meet the hierarchical latency guarantee like HQoS[84], they inspire us to propose the HBS to break the undifferentiated execution in polling mode and schedule VMs' batch I/O tasks on the limited IO-dedicated CPU cores hierarchically.

The main goal of HBS is to achieve latency differentiation. In the field of task scheduling, we have chosen a useful scheduling mechanism — the priority queuing, which has been widely used in operation systems and traffic control in datacenter[85–88]. The batch processing task in our environment is to forward packets from VM to NIC, so using a priority queue to reduce the waiting delay of high-priority tasks is equivalent to reducing the delay of packets inside them. Next, we introduce the HBS design in detail.

3.4.2.1 Class-based priority queues

The HBS design is shown in Fig. 3.8. To achieve hierarchical latency guarantee in HBS, as shown in Fig. 3.8(a), VMs are classified into 8 classes according to their priorities. The priority of each VM is determined by CSPs, e.g. the latency-sensitive services such as web or video can be set higher priority. These priorities will affect the order in which they are scheduled to perform batch I/O processing. On the data path shown in Fig. 3.8(b), all VMs' virtual structures are placed in virtual queues, and there are two kinds of queues: waiting queue and ready queue. As the C2TB allows the CPU to skip VMs with tokens less than 0, we put these VMs that should be skipped into the waiting queue. The VMs with tokens greater than 0 are queued in the corresponding ready queues according to their classes. The IO-dedicated CPU cores will only poll and dequeue the VM structures in the ready queues and do batch I/O forwarding tasks.

In the ready queues, to ensure that VMs in higher priority queues have lower latency, the higher priority queue has absolute execution privileges than the lower priority queue. For a queue with priority N, it will be executed dequeue operations only when there are no items in all $N-1$ queues with higher priorities. So in this case, the PMD threads poll each queue and perform



(a) Class-based arch. (b) The IO-dedicated CPU core processing logic under HBS

Figure 3.8 The hierarchical batch scheduling mechanism

batch execution according to the priority level. For example, in the case shown in Fig. 3.8(b), although the number of tokens in VM2 is the smallest among the VMs in the ready queues, VM2 will be dequeued and forwarded one batch of packets firstly because it has the highest priority. After the batch processing, VM2 is placed into the waiting queue for it has consumed 41000 tokens and its available tokens are negative. To ensure fairness that VMs in the same queue have similar latency, each virtual queue in the HBS follows the first-in-first-out (FIFO) policy.

3.4.2.2 Worst latency

With hierarchical execution privileges, the worst latency of VMs in each queue can be guaranteed and calculated. Although it is almost impossible to guarantee a specific value of the latency for each VM in software forwarding due to the uncertain hardware processing capacity, we still can guarantee the worst latency of VMs in each queue under HBS. We assume a case that the number of VMs in all 8 ready queues is $\{N_1, N_2, N_3, \cdots, N_8\}$, respectively. The time used for one batch processing is c. So the worst-case latency of VMs in these queues is $\{N_1 * c, (N_1 * k_1 + N_2) * c, (N_1 * k_1 + N_2 * k_2 + N_3) * c, \cdots, (N_1 * k_1 + N_2 * k_2 + \cdots + N_7 * k_7 + N_8) * c\}(k_i \geq 1)$. The variable k_i we use in this formula means that when a low-priority VM sends out one batch of packets, the higher-priority VMs may send out several batches of packets. So compared with original sequential execution that each VM equally suffers the worst $\sum N_i * c$ latency, HBS can provide hierarchical worst latency guarantee for VMs with different requirements. That helps CSPs formulate more flexible SLA policies based on the tenants' latency sensitivities.

3.4.2.3 Starvation avoidance

The last but most common problem in the HBS is how to avoid tasks starvation, which is the inherent problem in priority-based scheduling mechanisms. In HBS, starvation will occur in two situations: 1) The first one is that the low-priority VMs cannot achieve their purchased bandwidths when we prefer to use many resources to forward traffic for VMs with higher priorities. But in fact, when CPU resources are strictly allocated and isolated in C2TB, there is no case that the bandwidth of low-priority VMs is squeezed by others. 2) The second one will happen when many high-priority VMs have no traffic, but the CPU will still give priority to them and consume the allocated cycles. In this case, the low-priority VMs can get better network performance, but they still need to wait for CPU cores wasting time on the idle VMs. To avoid this, we allow the HBS to hold a dynamic priority for each VM. When a VM has no traffic to send during several consecutive batch I/O processing loops, the priority of this VM will be gradually dropped. But once it is found that the VM sends traffic again, it will be directly adjusted to the original priority.

3.5 Implementation

According to the design in Section 3.4, we implement the C2QoS strategy in the OVS-DPDK platform. As shown in Fig. 3.9, We modified the PMD thread's main loop function and the original port ingress policy, which are implemented by srTCM in the *ovs-vswitchd* module[67]. For each PMD thread, it has an independent HBS module to manage several VMs that it needs to be responsible for packet forwarding. Different PMD threads will not have access to each other's priority queues, so there are no competition and lock issues. The original sequentially polling running mode in the main loop of PMD threads is replaced by HBS that finds VM with the highest priority in the ready queues to execute batch I/O processing. The batch I/O processing logic of PMD threads has not changed, but before each batch processing, the rate limit strategies on VM ports are replaced with C2TB.

In order to make it easier for the CSPs to configure the VM's rate limiting and scheduling parameters on the command line, we also add two new commands to the OVS-DPDK. These commands realize our C2TB and HBS configurations through the *ovs-vsctl* module as shown in Fig. 3.9. For example, we can use command "*ovs-vsctl set interface vhost-user-1 ingress_policing_cpucycles=10000*" to allocate VM1, connected to "vhost-user-1" port, with 10000 CPU cycles per second for packet forwarding. The command "*ovs-vsctl set Interface vhost-user-1 options:priority=1*" can set VM1's priority as 1 in HBS scheduling. It should be

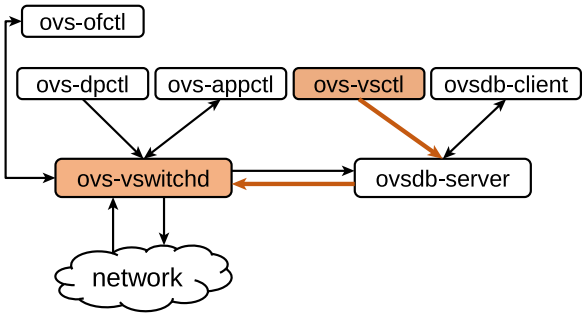


Figure 3.9 The modifications to original OVS-DPDK

noted that the C2QoS is a kind of general strategy, and in our implementation it can be used on all kinds of virtual ports such as "*vport*" and "*dpdk*" ports. All of these modifications require no more than 300 lines of code, which is easy to realize and meanwhile will not affect the original functions.

The biggest challenge in realizing C2TB is that it requires frequent measurement and calculation of CPU consumption in packet forwarding procedure, which will cause a great impact on vSwitch's forwarding performance without lightweight implementation. We use the *rdtsc*[89] instruction to solve it. The instruction *rdtsc* is to get CPU cycles from booting by reading the value in registers, so it has almost no overhead and can be widely used in data path. Another overhead comes from maintaining queues in HBS, and is undertaken by another manager thread. If it runs in busy polling mode, our strategy will consume one more entitle CPU core than the native OVS-DPDK, which is unacceptable. So we set the manager thread to be woken up every 50us to update the token number of each VM. The wake-up interval is a kind of trade-off that can be set to meet different needs. For example, setting a longer interval can reduce the additional CPU usage but will face a decrease in scheduling accuracy, while a smaller interval will consume more CPU resources. In our implementation here, we set it as the average time used for one batch I/O processing task to achieve the trade-off between accuracy and CPU consumption. The effectiveness and overhead of C2QoS will be evaluated in Section 3.6.

3.6 Evaluation

The main contribution of C2QoS is to ensure the bandwidth and latency of VMs on the physical server, so in this section, we evaluate the VM network QoS guarantee under C2QoS and the OVS-DPDK existing "ovs-ingress-policy" QoS strategy. Our experiments include the following aspects:

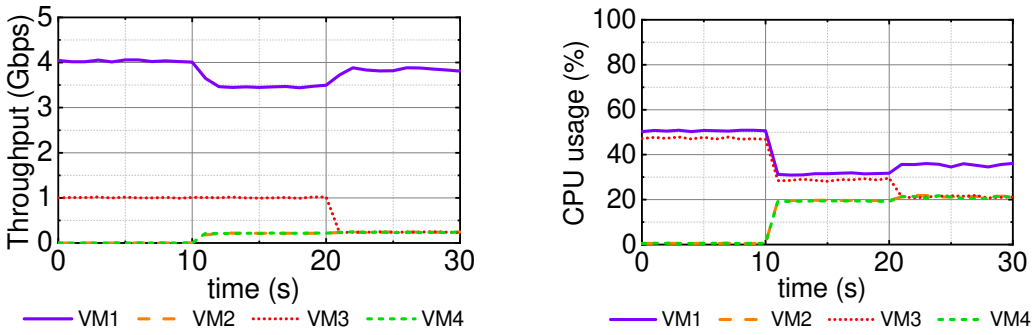
- Bandwidth and latency guarantee tests: Comparing the VM's TCP bandwidth and latency guarantee of C2QoS with that of ovs-ingress-policy.
- Accuracy: Measuring the accuracy of rate limiting in C2TB, and the latency levels of different priority queues in HBS.
- Application experiments: Comparing the throughput of the Ftp[90] server and response latency of the Nginx[91] server under the C2QoS and ovs-ingress-policy.
- CPU overhead: Measuring the additional CPU overhead that C2QoS brings to OVS-DPDK.

The hardware and platform configurations are the same as described in Section 3.2.2.

3.6.1 TCP bandwidth and latency

In this experiment, we use iperf[92] and qperf[77] tools to evaluate VMs' TCP bandwidth and latency. We launch 4 VMs with 4 Gbps, 4 Gbps, 1 Gbps, and 1 Gbps purchased bandwidth, respectively, and use one dedicated CPU core in OVS-DPDK for forwarding. The purchased bandwidths are guaranteed based on the SLA that sets the preferred packet size to 1024 bytes and the number of flows to 1. According to our pre-measured model (as described in Section 3.3), when deploying 4 VMs on NUMA node 1, the VMs with (4 Gbps, 1024-byte packet, single flow) need 0.792G cycles/s (36%) CPU resources for packet forwarding, and the VMs with (1 Gbps, 1024-byte packet, single flow) need 0.198G cycles/s (9%) CPU resources.

In our benchmark setting, VM1 acts as a well-behaved tenant and sends 1024-byte packets at the maximum rate all the time while the other 3 VMs behave as malicious users or noisy neighbors. In order to more clearly show the difference between C2TB and the original rate limiting method in OVS-DPDK, we design such a circumstance: 1) In the first 10 seconds, VM1 and VM3 send traffic with 1024-byte packets. The VM2 and VM4 are sleeping and generate no traffic. 2) From 10th second to 20th second, VM1 and VM3 keep sending traffic with 1024-byte packets, while VM2 and VM4 send traffic with 64-byte packets. 3) In the last 10 seconds, VM1 still sends traffic with 1024-byte packets, while VM2, VM3 and VM4 send traffic with 64-byte packets at the maximum speed. The TCP bandwidth and CPU consumption of the 4 VMs under three strategies are shown in Fig. 3.10, Fig. 3.11 and Fig. 3.12. In these figures, "C2TB-strict" means we only assign each VM the fixed C_{alloc} for packet forwarding, and the "C2TB-MINMAX" supports to entirely allocate C_{idle} and unused CPU resources to VMs for completing MIN-MAX bandwidth allocation.



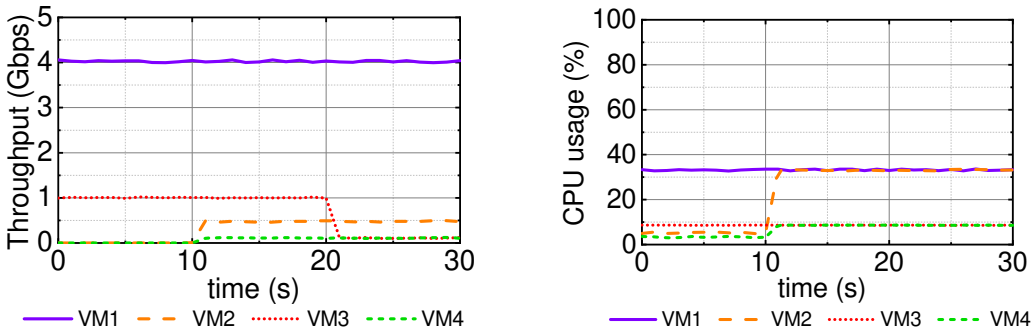
(a) TCP bandwidth

(b) CPU usage

Figure 3.10 The VM network bandwidth and CPU usage under "ovs-ingress-policy"

Fig. 3.10 shows the results under default ovs-ingress-policy. It can be seen that VM1 and VM3 work well and keep their purchased bandwidth within the first 10 seconds. But from the 10th second, VM2 and VM4 start sending 64-byte packets, and they both compete and occupy 20% of the CPU resources. As shown in Fig. 3.10(b), the behavior of VM2 and VM4 severely squeezes the CPU resources that were originally used by VM1 and VM3. As a result, VM1 bandwidth drops by 12%. But for VM3, although the available CPU resources of VM3 have been squeezed, they are still enough to support VM3's purchased bandwidth. In the last 10 seconds, an interesting thing comes that the bandwidth and CPU usage of VM1 increase with the VM3 changes packet size from 1024 bytes to 64 bytes. But VM3 does not benefit from the change of traffic characteristics. We print all the log information and find this is caused by the running mode of OVS-DPDK. The sequential execution in the PMD thread makes it equal in the number of batch I/O processing loops performed for each VM per second. As VM3 reduces the packet size, the number of batch processing loops of each VM per second is increased. For VM1, the increase in the number of batch processing loops per second means that more packets can be sent per second (before the bandwidth reaches the rate limiting threshold), which increases bandwidth and CPU consumption. So under the OVS default BPS-based rate limiting strategy, the behavior of the tenants will cause unpredictable CPU allocation, and cannot guarantee VM bandwidth.

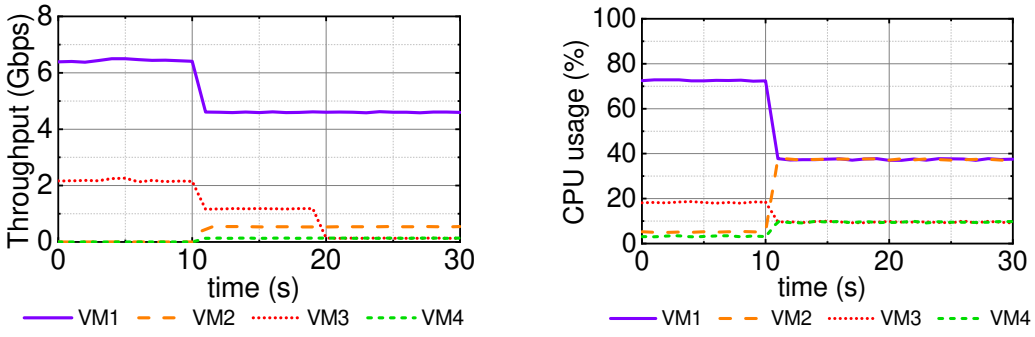
The bandwidth and CPU consumption of each VM under the C2TB-strict strategy are shown in Fig. 3.11. According to the modeling results, the CPU resources that we should allocate to the 4 VMs are 36%, 36%, 9% and 9%, respectively. We analyze the bandwidth of each VM separately. For VM1, since its behavior keeps unchanged, its bandwidth is stable and keeps at 4 Gbps by using 36% of the CPU resources for forwarding all the time. For VM3, changing the packet size to 64 bytes in the last 10 seconds can only reduce its own bandwidth. For the two attackers (i.e. VM2 and VM4), in the event of sending 64-byte packets, the CPU resources



(a) TCP bandwidth (b) CPU usage

Figure 3.11 The VM network bandwidth and CPU usage under "C2TB-strict"

allocated to them can only achieve very low bandwidth, and they cannot interfere with other VMs by competing for more CPU resources. It should be noted that in the last 10 seconds, VM3 and VM4 get extremely low bandwidth, and it seems they face starvation. But in fact, that is exactly what we want to achieve. The problem of bandwidth isolation is caused by these VMs using special traffic characteristics, rather than their preferences, to compete for more IO-dedicated CPU resources. The solution in C2TB is to let these "noisy" VMs only affect their own network performance by restricting CPU consumption for each VM. Compared with the ovs-ingress-policy, CBTB-strict can guarantee well-behaved VMs' bandwidth and eliminate the CPU resources competition.



(a) TCP bandwidth (b) CPU usage

Figure 3.12 The VM network bandwidth and CPU usage under "C2TB-MINMAX"

While providing good isolation, the C2TB-strict still causes a waste of CPU resources in the vSwitch, as shown in Fig. 3.11(b), nearly 75% of CPU resources are wasted in the first 10 seconds. So the C2TB-MINMAX is used to solve this kind of waste. Comparing Fig. 3.11(a) and Fig. 3.12(a), it can be seen that the main difference between C2TB-MINMAX and C2TB-strict happens in the first 10 seconds. With C2TB-MINMAX, VM1 and VM3 make full use of

all idle CPU resources on the server and achieve higher bandwidth (6.3 Gbps and 2.1 Gbps) than the bandwidth they purchased (4 Gbps and 1 Gbps). So this rate limiting method also has good robustness while guaranteeing the network QoS of VMs.

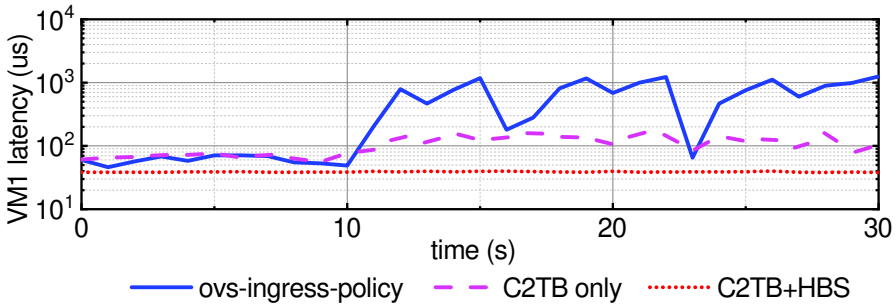


Figure 3.13 The experimental results of TCP latency

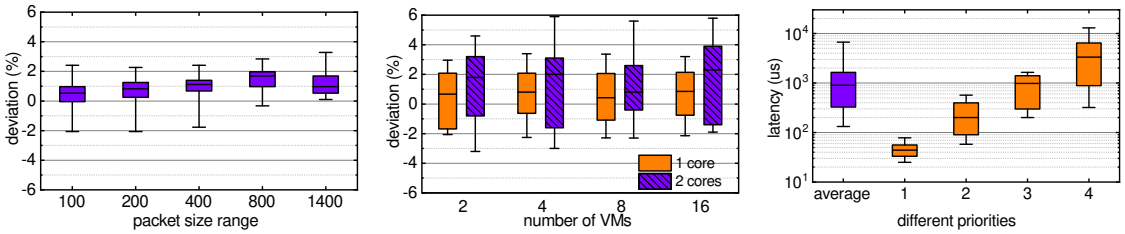
On the aspect of TCP latency, we separately use qperf to measure the VM1 latency under the ovs-ingress-policy, "C2TB only", and C2TB+HBS strategies. The results are shown in Fig. 3.13. Under the ovs-ingress-policy, in the first 10 seconds, only VM3 competes with VM1 for the CPU core to do batch I/O processing tasks, which leads to a slight increase in VM1 TCP latency. In the following 20 seconds, the latency of VM1 becomes unstable and increases significantly (more than 1ms in the worst case) due to the competition of the other three VMs. Compared to ovs-ingress-policy, "C2TB only" can reduce part of the additional latency of VM1 by skipping ports with negative tokens. Another reason for the lower latency under C2TB is that it keeps the packets not being sent inside the VM, forming a "back-pressure" to the senders and adjusting the sending rate of the TCP protocol stack in VMs. But the latency under "C2TB only" is still unstable. With HBS, we set VM1 to be placed in the Priority 1 queue which ensures VM1's forwarding tasks always to be executed first. The results show that the VM1 latency under C2TB+HBS is close to the native performance and is not affected by other VMs.

Therefore, with these experiments, C2QoS can provide good isolation from the CPU level. That enables C2QoS to provide tenants with good network SLA performance guarantees under the conditions of CPU resources competition and variable processing capacity in vSwitch.

3.6.2 Accuracy

In addition to the advantages on isolation, we also need to evaluate the accuracy of C2QoS. Based on the functions of C2TB and HBS, the accuracy is reflected in two aspects: the accuracy of rate limiting and the hierarchy of the worst latency.

We first evaluate the deviations of C2TB under mixed-size packets and the results are shown in Fig. 3.14(a). In this experiment, we use pkt-gen in VM to send packets with mixed sizes but keep a fixed average size. Then we set the average packet size parameter for C2TB according to the modeling methodology in Section 3.3.3. We can see that the range of the packet size has little effect on the accuracy. Most of the results show that the deviation is greater than 0, which means that in most cases, we can guarantee that the VM's available bandwidth is greater than or equal to its purchased bandwidth. On average, the deviation of C2TB under mixed packet size is between (-2%, 3%).



(a) Mixed-size packets (b) Deployment (c) Hierarchical latency

Figure 3.14 The accuracy evaluation

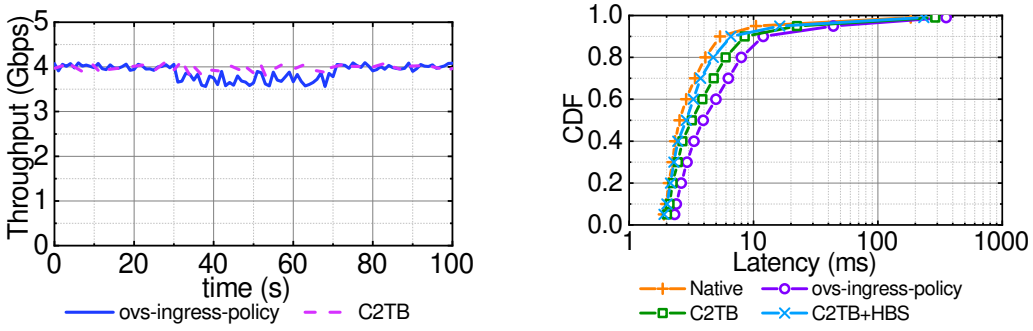
In Fig. 3.14(b), we increase the number of VMs and deploy them on every NUMA node to compare their real bandwidth with their purchased bandwidth. The increase in the number of VMs did not have large impacts on the deviation. But it can be concluded that the more variables introduced in the modeling, the greater the deviations are. When using only one CPU core for forwarding in vSwitch, the deviation is between (-2%, 4%). But in the case of using two cores for forwarding, resource competition becomes even more unpredictable, so the deviation has almost doubled to (-3%, 6%). Although the accuracy of the C2TB is incomparable to the traditional precise rate limiting methods, the CSPs believe the rate limiting under software forwarding does not need to be so precise and occupies many resources[15]. So the deviations of C2TB are acceptable.

To evaluate what kind of latency levels can HBS provide, we run 16 VMs belonging to 4 priorities on one dedicated CPU core for forwarding, and evaluate their TCP latencies under the case that all VMs are sending traffic concurrently. The results are shown in Fig. 3.14(c). The "average" in this figure is the average latency of all 16 VMs under the C2TB mechanism only. In this case, since all VMs need to wait for batch I/O processing, their latencies are high and unstable. In HBS mechanism, we can see that although the latencies of VMs in different priority queues have intersections, the latency levels in most cases are obviously different. The latency of VMs with priority 1 and 2 is less than average latency, while the latency of VMs with priority 3 and 4 is much worse than the average. Another fact is that the latency

distribution of high-priority VMs is very concentrated. But as the VM priority decreases, the frequency of these VMs' batch I/O processing will be more uncertain, which contributes to the high discreteness. The different latency levels brought by HBS will be useful when providing differentiated services for tenants.

3.6.3 Application results

To make it more practical, we consider some common applications on the public cloud. For example, the latency-sensitive VMs (such as website and video services) compete with bandwidth-sensitive VMs (such as online disks) for vSwitch forwarding resources on the same physical server. So we evaluate the bandwidth of the Ftp server and response latency of the Nginx server in this experiment. We choose Nginx not only because it is a latency-sensitive service, but also because of its special traffic characteristics. The traffic of Nginx is usually composed of small packets, and the five-tuple segments of these packets are discrete in the case of high concurrency. That will cause more CPU consumption in *classification* stage (as described in Section 3.3.2). For test configurations, 2 VMs with 4 Gbps bandwidth are deployed as Ftp servers and 2 VMs with 1 Gbps are deployed as Nginx servers. The Ftp servers keep sending traffic while the Nginx servers bear pressure test during 30th-70th seconds using the wrk[93] tool.



(a) Ftp bandwidth

(b) Nginx response time

Figure 3.15 The performance results of applications

The Ftp bandwidth is shown in Fig. 3.15(a), the Nginx servers' concurrent traffic during 30th-70th seconds causes a bandwidth drop of about 11% on the Ftp servers under ovs-ingress-policy, while C2TB strictly guarantees the bandwidth of Ftp servers all the time. For the latency in the Nginx pressure test, we obtain the request response time distribution in Fig. 3.15(b). Under ovs-ingress-policy, the response time of Nginx requests is doubled compared to native performance. When only using C2TB, 50% additional latency is reduced by skipping ports with

tokens less than 0. But with the C2QoS containing both C2TB and HBS, the additional latency is reduced by more than 80% and these Nginx servers achieve almost the native performance. Therefore, the C2QoS can ensure the network performance of both latency-sensitive and bandwidth-sensitive services while sharing the same physical resources in vSwitch.

3.6.4 Overhead

As we added a new module to vSwitch, it may bring new overhead. The overhead mainly reflects on two aspects: the performance decrease and the additional CPU overhead.

For the first concern, in the single-VM and multi-VM experiments, the OVS-DPDK using C2QoS strategy has no performance drop compared with the original version. This is because we have not made big changes to the data path, and the additional function added to PMD threads only contains CPU cycles counting. The CPU cycles counting function is composed of *rdtsc* instruction[89], which occupies only several cycles and has very little effect on forwarding performance.

For the additional CPU overhead, we tested the CPU consumption of C2QoS. On the IO-dedicated CPU cores, only 0.018% of the CPU usage is used for C2QoS. This part of CPU usage will not go up with the increase in the number of VMs because it is added in the batch I/O processing of all VMs' ports. Besides the overhead on IO-dedicated CPU cores, the manager thread's CPU overhead also needs to be considered. When deploying 28 VMs, 2.08% more CPU usage is used for token counting and queue managing. Moreover, the CPU usage of the manager thread can also be reduced by sacrificing accuracy and extending the wake-up interval. Although it is a trade-off, from our experimental results, the additional CPU resources consumed by the manager thread will not be too much (no more than 3%). So the additional CPU overhead in C2QoS is also acceptable for cloud platforms.

3.7 Conclusion

This chapter focuses on VM network performance isolation on the cloud platform, and solves the key problem that existing QoS methods ignore I/O-dedicated CPU resource competition in vSwitch. Specifically, the VMs' network forwarding tasks compete for limited CPU resources in vSwitch in terms of utilization and timing, which seriously affects tenant network experience and cannot guarantee stable network QoS. To resolve the issue, we proposed C2QoS to

apportion and schedule IO-dedicated CPU resources to VMs for network SLA guarantee. C2QoS consists of two mechanisms, C2TB and HBS. In C2TB, according to a measurement-driven bandwidth-CPU model, we limited VM's bandwidth by directly assigning CPU cycles to a particular VM. To address the high additional latency issue brought by the undifferentiated execution, the HBS mechanism scheduled the VMs' entire batch I/O forwarding tasks on the IO-dedicated CPU cores, which provided hierarchical latencies for VMs according to sensitivities. The implementation on the OVS-DPDK platform showed that compared with existing strategies, C2QoS eliminated the influence of CPU resource congestion on bandwidth and reduced the effect on latency by 80%.imanyis

Chapter 4

D-TSE: Flow Table Isolation based Data Plane Attack Defense Mechanism

4.1 Introduction

Public cloud is becoming a popular trend as it allows tenants to deploy services flexibly in the form of VMs[9]. To realize the network connectivity between the VMs and external devices, vSwitch is introduced to implement packet classification and forwarding for the multiple VMs on the physical server[69, 64]. As a software process that uses CPU for packet forwarding, the flexibility and performance of vSwitch have been concerned for a long time.

In existing vSwitches, SDN is widely supported to solve these concerns. With the "match + actions" abstraction, vSwitch can separate the packet matching from the actions to be performed on the packets. So the various types of tables in vSwitch, such as routing table, ACL table, NAT table, etc., can all be replaced by a unified flow table. At the same time, to improve efficiency and maximize performance, all tenants in vSwitches share the key data structure and processing logic. In this way, each packet can be quickly matched in the centralized flow tables to determine the actions to be executed and whether the packet should be sent to the SDN controller for "pulling" new rules.

However, the cost is that the vSwitch will face the risks of isolation breakage and DoS attacks. Since all the tenants share data structure and CPU resources for packet classification, their network performance can easily be affected. That provides an attack surface for the malicious tenants. A large number of studies in the last ten years have shown that the performance of packet classification in vSwitch can be reduced to an extremely low level to exhaust CPU

resources by sending specific traffic with carefully designed packet header fields[24–27, 94]. But most of the preconditions of these attacks are not realistic enough. For example, they require to send malicious traffic at a very high rate or need to know the rules in the SDN controller in advance.

Recently, Csikor et al. make it more practical to realize. They proposed an attack called TSE on the OVS-DPDK[59, 11] that only requires low-rate traffic with arbitrary packet header fields[28]. The malicious traffic will "pull" numerous useless rules from the SDN controller to the shared flow tables in OVS-DPDK, and greatly increase the time/space complexity of the TSS packet classification algorithm. As a result, the CPU resources are short and all tenants suffer from low network performance. That threatens the stability of cloud services and needs to be resolved urgently.

In this chapter, we first reproduce the TSE attack on the OVS-DPDK platform and analyze how the attack works. Different from some limited solutions focusing on a specific algorithm or system[95, 28, 30], we analyze the root causes of TSE attack and its premises from a more basic and general perspective. From the resources management in the whole vSwitch system, we find two premises to realize the TSE attack. The first one is the lack of isolation in the flow tables. As all the tenants share the same flow table structure for classification, the malicious tenant can easily change the time/space complexity of other tenants' packet classification. The second one is the absence of CPU isolation. Since the CPU resources that vSwitch can use are limited, malicious tenants can increase their classification complexity and compete for most of the IO-dedicated CPU resources, which will also leads to a DoS attack.

Based on these two attack premises, we propose D-TSE, the defense strategy to protect the innocent tenants from TSE attack. D-TSE is also an algorithm/platform independent design principle, which can guide the design of vSwitches to avoid possible attack risks. In order to achieve this goal, this chapter makes the following contributions:

- We propose a tenant-level flow tables isolation mechanism. By adding a lightweight PRECLS classifier to distinguish and redirect traffic to each tenant's own flow tables for stateful packet matching, each tenant enjoys independent lookup time/space complexity.
- We present a fine-grained CPU resources isolation mechanism for tenants. We limit the CPU cycles that each tenant's forwarding tasks can consume per second to enhance the physical resource isolation.
- We implement the D-TSE strategy on the OVS-DPDK platform. The experiments show that it defends against TSE attack at the cost of less than 5% performance drop.

4.2 Background and motivation

As an indispensable component of network virtualization, the vSwitch takes the task of providing network connectivity for VMs, while supporting a series of features like subnet isolation and network discovery in the cloud network. Although the services provided by vSwitches are becoming extremely complex under the current cloud scale, the main logic of packet processing remains unchanged.

The packets received from the NIC will go through *ingress*, *packet classification* and *egress* (packet copying between host and VM memory) procedures to be obtained by the VMs, and vice versa. Among them, the operations performed by the vSwitch in the two stages of *ingress* and *egress* are very simple and usually done by the driver, while the *classification* stage contains complex "match + actions". Therefore the performance during *classification* stage is easily affected, which makes vSwitch vulnerable to DoS attacks.

4.2.1 Packet classification in vSwitch

Since OVS-DPDK is the most widely used open source vSwitch and represents the mainstream design, we first use it to analyze the packet *classification* in vSwitch.

The workflow of packet *classification* in OVS-DPDK is shown in Fig. 4.1. According to the workload, the CSPs can launch one or several PMD threads in OVS-DPDK and bind them to dedicated CPU cores. These PMD threads receive packets from the VM ports or NIC ports that they are responsible for, then classify and execute *actions* for these received packets. In order to improve performance and enhance scalability, the packet classification in OVS-DPDK adopts a three-level classifier structure which is composed of Exact Matching Cache (EMC), Megaflow cache (MFC) and OpenFlow classifier. The search performance in each level classifier is about $O(1) : O(10) : O(100)$[59]. Each PMD thread maintains only one EMC classifier regardless how many ports it needs to poll, and each port has its own MFC in the PMD thread. So tenants inevitably share these classifiers/flow tables. We will introduce briefly how a packet lookup is performed in these classifiers.

When a packet is received from a port, it will be firstly sent to EMC for packet matching. Each PMD thread maintains only one EMC classifier with 8192 entries by default, so it can be said that all tenants share the EMC storage space for these 8192 rules. As an accelerator for packet classification, EMC's design is very simple. The entire EMC is implemented as a hash table, so that PMD thread only needs to perform one hash lookup for each packet to determine whether

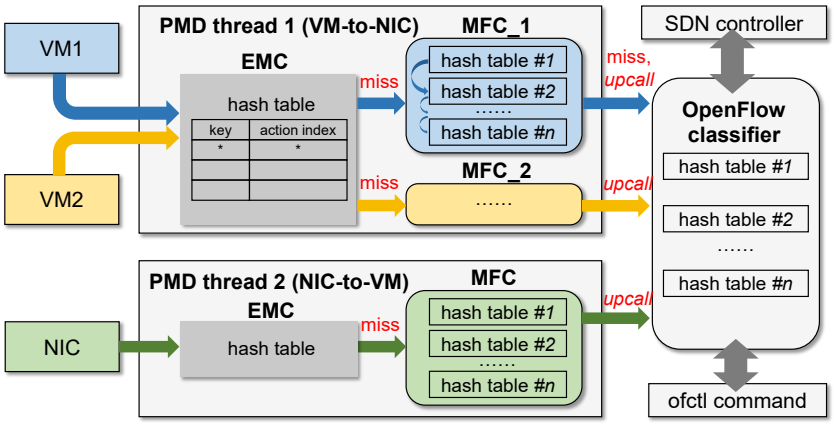


Figure 4.1 The three-level classifier structure in OVS-DPDK

matched or not. A hit in the EMC can end the matching process and speed up the packet processing.

If the packet fails to get matched entry in EMC, it will be continue to match in its corresponding MFC based on its received port. Each MFC corresponds to a distinct port. That means from the direction of VM-to-NIC, each VM has its own independent MFC classifier, but all VMs share one MFC in NIC-to-VM direction (see Fig. 4.1). By adopting TSS algorithm, MFC contains many subtables to support wildcard matching. The rules with the same prefix length are stored in the same subtable and there is none overlapping rules. For packet matching, the PMD thread will perform a hash lookup in each subtable in sequence. Considering the all possible prefix length in $\langle dst\_ip, dst\_port, src\_ip, src\_port \rangle$, there could be up to $32 * 16 * 32 * 16 = 262144$ subtables in MFC and 262144 hash lookup in the worst case. So it can be seen that the number of subtables has huge impacts on the MFC lookup performance.

When the packet matching missed in both EMC and MFC, the PMD thread will send an "upcall" to the OpenFlow classifier and specific thread will handle it. The OpenFlow classifier stores the original rules came from the controller and CSP's command line input. With the inter-thread communication and unoptimized structure, it has a poor lookup performance. After the packet finally get matched entry in OpenFlow classifier, the matched rule will be inserted into EMC and MFC. That means the traffic can change the number and distribution of rules in the first two level flow tables.

Through the above analysis, we can see that the traffic can change the number and distribution of rules in the first two level of flow tables by "pulling" rules from the OpenFlow classifier. This is an important guarantee for simplifying network configuration and enhancing flexibility under SDN. But considering the fact that these classifiers/flow tables are shared among tenants (e.g.

see Fig. 4.1, the EMC in VM-to-NIC direction, both EMC and MFC in NIC-to-VM direction), this feature will also bring the opportunity for attacks.

4.2.2 TSE attack and existing defense mechanisms

By exploiting the feature in SDN that traffic can "pull" rules from controller and insert them into the shared flow tables, some works have already warned that attackers can used well-designed traffic to trigger DoS attacks[24]. They can cause innocent tenants suffering from frequent packet matching failures through "pulling" a large number of useless rules to the shared flow tables. But most of these works are far from real environment, because they require high rate traffic or need to know existing rules in SDN controller in advance.

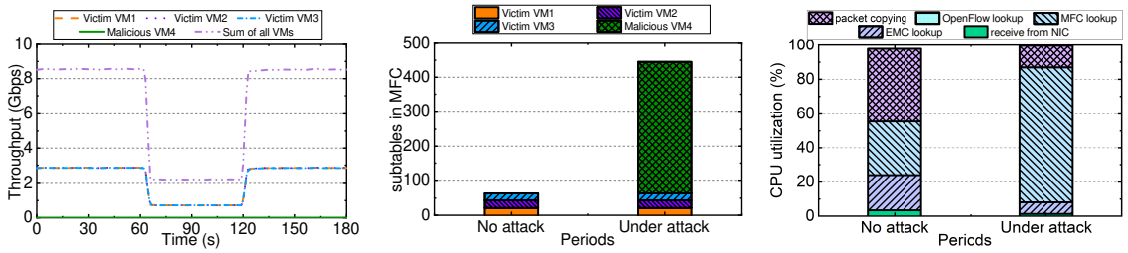
Recently, a practical attack called TSE is proposed and implemented on the popular OVS-DPDK platform[28]. The prerequisites of TSE attack are close to the real production environment. The attacker only needs to send low-rate traffic with arbitrary packet header fields to achieve DoS attack on OVS-DPDK. The malicious traffic first "pulls" rules to exhaust the space in EMC and then inserts as much as possible useless rules with different prefix length into MFC from the OpenFlow classifier. So the malicious traffic greatly increases the time complexity of packet matching in these shared classifiers. None of the other VMs deployed on this physical server are immune to the dramatic decrease in network performance.

We use experiments to reproduce and analyze the TSE attack. In this experiment setting, we run 3 VMs as victims (with 3 Gbps purchased bandwidth) and 1 VM as the malicious tenant (with 1 Gbps purchased bandwidth). We use the testcenter from Spirent[96] as the traffic generator, and run DPDK l2fwd program inside these VMs to forward traffic back. In addition to the basic routing rules, we add 2 ACL rules[1] for VM4 in the OVS-DPDK with a prefix length of 24. According to the methods in [28], up to $24 * 24 = 576$ subtables with different prefix lengths can be generated in MFC through "pulling" rules. Then we use testcenter to send normal traffic with only randomized src_ip to the 3 victim VMs all the time, and send malicious traffic with randomized src_ip, dst_ip, src_port and dst_port to VM4 during the 60th-120th seconds.

The throughput results of all VMs during 0-180th seconds are shown in Fig. 4.2(a). It is obvious that all 3 victim VMs suffer from 70% bandwidth drop during the TSE attack period (60th-120th seconds). The reason of the network bandwidth drop can be found in Fig. 4.2(b) and Fig. 4.2(c). Under TSE attack, the low-rate malicious traffic firstly "pulls" rules to exhaust all the 8192 entries in EMC and then increases the number of subtables in MFC. On average,

[1] The public CSPs all allow tenants to add their own ACL rules to the vSwitch[69, 64]

the traffic from VM4 contributes nearly 400 subtables in the shared MFC of PMD thread 2, while traffic from the other 3 victim VMs only add 60 subtables. With the increase of subtables in MFC, the CPU resources of PMD thread 2 spent on MFC lookup also increase significantly. As shown in Fig. 4.2(c), the CPU consumption of the MFC table lookup under the TSE attack increases by more than 2 times, and occupies 78% of the whole CPU core.



(a) Throughput of all VMs (b) The number of subtables (c) The CPU usage

Figure 4.2 The reproduction and analysis of TSE attack

Some intuitive and simple mechanisms are proposed to detect malicious traffic and limit the frequency of *upcalls*[26–28]. However, these methods can only reduce the effect of TSE attack, but cannot fundamentally solve this problem. To make matters worse, normal traffic will be treated as malicious traffic in some cases. For example, for the website services, the *src_ip* and *src_port* of its packets are also discrete. These simple defense mechanisms will harm the QoS in these scenarios.

4.2.3 Motivation

From our views, the feature in SDN and the classification algorithm should not be blamed. In fact, TSE attack represents a type of attacks that exploit the lack of isolation in the flow tables of vSwitch, and that challenges the stability of public cloud. With the design that tenants share flow tables, malicious tenants have the chance to change the time/space complexity of other tenants' packet classification. As shown in Fig. 4.2(b) and Fig. 4.2(c), the useless rules "pulled" by the malicious traffic can increase the time complexity of packet matching and consume most of the IO-dedicated CPU resources. As a result, the forwarding capacity of vSwitch drops to an extremely low level, and a DoS attack is formed.

In this chapter, we want to design strategy to improve the isolation among tenants in terms of flow tables and CPU resources, so as to thoroughly prevent TSE and other attacks based on isolation breakage. To isolate flow tables in NIC-to-VM direction, the challenge comes that how to break the paradox to determine the destination VM of each packet before the

classification stage. For CPU resources isolation, the challenge is how to achieve fine-grained CPU consumption limitation. In the next section, we will describe our solutions.

4.3 Design

To prevent TSE attack, D-TSE strategy should achieve tenant-level isolation on two aspects: isolating flow tables; and isolating the IO-dedicated CPU resources. We will separately illustrate in detail how we solve the challenges to achieve this two types of isolation. Although we take OVS-DPDK as an example, the design principles are platform independent and can be easily applied to other vSwitches.

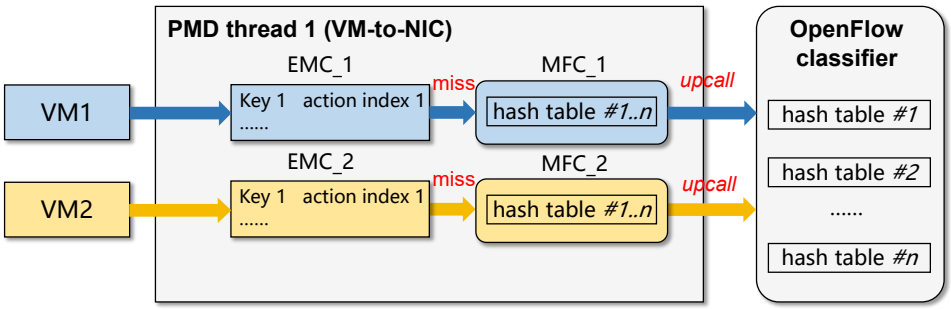
4.3.1 Separated flow table structure

This subsection will show the proposed split flow table design to provide independent table lookup performance and fault point isolation, and the proposed early classification design to break the paradox.

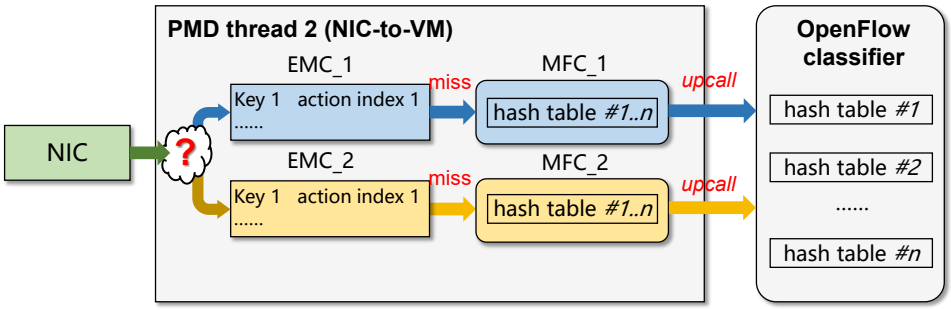
4.3.1.1 Separated three-level flow table

To isolating the possible common failure point, the shared data structures must be isolated for tenants in the vSwitch. Specifically, we need to isolate the three-level flow table/classifier at the granularity of tenant in vSwitches. As can be seen in Fig. 4.1, the shared flow table in native OVS-DPDK includes: 1) EMC in each PMD thread; 2) MFC in the NIC-to-VM direction. For the OpenFlow classifier, its main role is to store the original rules from the controller, and its I/O-dedicated CPU consumption is negligible even under TSE attack (see Fig. 4.2(c)), so it is not necessary isolate it in this mechanism design.

Fig. 4.3(a) shows the design of the separated flow table in the direction of the VM-to-NIC. Since the packets from different tenants can be judged by the receiving ports, it is easy to realize the isolation. In this direction, there is no isolation issue in MFC because each port corresponds to one dedicated MFC classifier. For EMC, a separated EMC needs to be assigned to each VM port. In this way, each time the PMD thread receives a batch of packets from one VM port, it can directly send them to the private EMC and MFC classifiers for classification. That ensures each VM has its own isolated flow table structure along with the classification



(a) VM-to-NIC direction



(b) NIC-to-VM direction

Figure 4.3 The design of separated three-level flow table

performance. Besides, this change also makes it more convenient for CSPs to manage the flow table capacity in EMC and MFC classifiers for tenants.

In the direction of NIC-to-VM, we similarly assigns an independent EMC and MFC to each VM in Fig. 4.3(b), but encounters an actual issue. The PMD thread 2 only needs to receive packets from the NIC port, and these packets may belong to any VMs on the server. So it is impossible to directly judge the belongings of the packets according to the port before classifying them in the shared EMC and MFC classifiers. But on the other hand, if the packets go through the matching process in the shared EMC and MFC, then the isolation mechanism to defend against attacks is invalid. Therefore, that requires the design of a lightweight early classification mechanism to determine the belongings of packets and redirect them to separated EMC and MFC classifiers.

4.3.1.2 Early classification

In order to break the paradox on the premise of ensuring performance, we add PRECLS, a hash table based lightweight classifier, before the EMC matching process. The aim of PRECLS is

to quickly distinguish the belonging VM of each received packet, and redirect the packet to corresponding EMC and MFC for subsequent process.

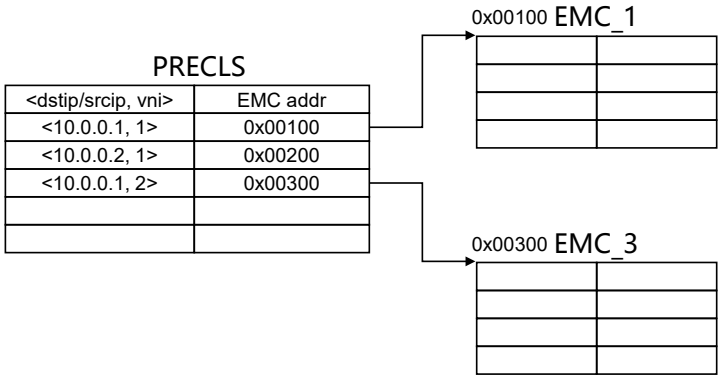


Figure 4.4 The design of PRECLS

Fig. 4.4 shows the structure of PRECLS. In the cloud environment, the table stored in the PRECLS is not a flow table, but a simple mapping table. In this mapping table, the key of each entry is $\langle dstip \rangle$ of the packet (the VM's IP adress), or add VXLAN Network Identifier (VNI) as $\langle dstip, vni \rangle$ when it is necessary. The value of each entry is the address pointed to the EMC classifier of the packet's belonging VM. When a packet arrives at the NIC, the PMD thread will first look up in PRECLS and find its private EMC classifier, and then redirect the packet to its private EMC classifier. The subsequent stateful packet matching and forwarding operations are performed separately for each VM's packets. The lookup process in PRECLS only consumes one hash lookup, and the entries in the entire table will not exceed the number of VMs deployed on the server. So it has little impact on performance.

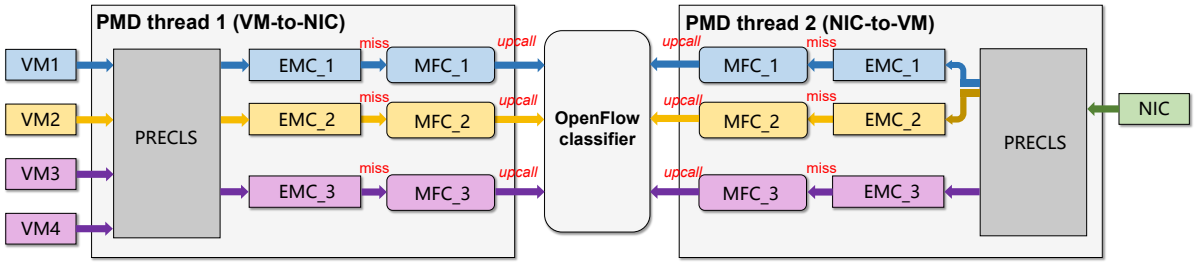


Figure 4.5 The isolation design for multi-level flow tables in D-TSE

For the consistency of the data plane logic, we also adds a PRECLS in the direction of VM-to-NIC. As shown in the Fig. 4.5, the entry's key in PRECLS adopts is the packet's $\langle srcip \rangle$ (the VM's IP adress). This can facilitate service fast discovery for tenants with multiple VMs. As can be seen from the purple part in the Fig. 4.5, the VMs belonging to the same tenant are combined into a "group", which is mapped to the same EMC and MFC classifiers through

PRECLS. In this way, if VM3 communicates with the outside devices and adds a new entry into PRECLS. According to the locality principle, it is very likely that VM4 owned by the same tenant will also use this entry in a short period of time, thus saving update time in VM4's flow tables.

The maintenance of the PRECLS is completed by the virtualization platform and the vSwitch. Each time when a new VM instance is deployed, the vSwitch will create a set of private EMC and MFC flow tables for it, and then its IP address will be inserted into the PRECLS table. When a VM instance is deleted or migrated to another server, its private flow tables will be destoryed and its entry in PRECLS will be deleted accordingly.

4.3.1.3 Rule update

Since the D-TSE mechanism changes the structure of the three-level flow table in the vSwitch, the rule update process will also be changed accordingly. Under the network abstraction of SDN, there are two situations for updating flow table rules: 1) The controller actively issues rule updates; 2) When the forwarding device receives a packet that fails to match in flow table, it will send a "packet_in" message to request the update. In the following, we describes the changes of these two update behaviors under the D-TSE mechanism.

For the active update from the controller, there are generally two cases. The first one is that the controller needs to update the rules in local flow tables of all forwarding devices due to changes in the global route and gateway IP in the cloud network. In the native OVS-DPDK, this update only requires one time of update in the shared flow table. But under the separate flow table structure of D-TSE, this update requires updates in all tenants' separated flow tables on the server. So the update cost of this part increases by n times (n depends on the number of VMs). Considering that this kind of update is infrequent, it has little impact on the vSwitch. The second case of update comes from the rules added by tenants, such as ACL rules and security group settings. In this case, the D-TSE mechanism does not increase the times of rule update, but only changes the rule update in the shared flow table into the rule update in the specific tenant's flow table.

For the rule update initiated by the forwarding device, it is strongly related to the behavior of the tenants. Because the resources that tenants access vary greatly, the flow entries required by vSwitch to forward packets are also quite different. Under this premise, the D-TSE mechanism will not introduce increases in the number of updates. In the native OVS-DPDK, each time tenants send traffic to access the external resources, the controller will be triggered to issue

rules and insert them into the shared flow table. After adopting the D-TSE mechanism, the change is just to select the corresponding tenant's flow table structure to update.

Except for selecting which flow table to update, other operations during rule update maintain unchanged compared to the native OVS-DPDK. This is due to that the D-TSE mechanism does not change the original look up algorithm, and the rule update operations in each separated flow table still follows the TSS algorithm.

4.3.2 Batch re-aggregation

Although the PRECLS does not introduce too much overhead, the packet redirection operations in NIC-to-VM direction will break the efficient batch processing, thereby drop the performance. This section describes how to address possible performance degradations, as well as the overhead analysis.

4.3.2.1 Aggregation process

In the native OVS-DPDK, the batch processing mode runs through the entire PMD thread: a batch of packets are received from the NIC port, and then this batch of packets enters the EMC, MFC and OpenFlow classifiers for matching. Finally, according to the matching results, this batch of packets will be sent to the corresponding destination ports. The high efficiency of batch processing comes from amortizing the overhead of redundant operations such as fetching addresses, and improving the cache hit rate. So the batch processing is extremely important for high-performance data path.

However, due to the introduction of the PRECLS, each batch of packets received from the NIC will be broken up and redirected to different EMC and MFC classifiers for matching and subsequent processing. If we continue to use the previous batch processing and perform stateful flow table matching for each packet, the cache line will inevitably be flushed and reloaded. Thus the processing performance will be degraded as the continuity is broken.

In order to solve this problem, we implements a batch re-aggregation mechanism. After the PRECLS matching is completed, the packets with same destination VM are re-aggregated to maintain the batch processing logic in the subsequent processing. As shown in lines 5-7 of the Algorithm 2, we reorganize the packets and divide them into an batch array. The size of the array is equal to the number of VMs. Then the PMD thread performs complex packet matching and forwarding for each batch in this array, which ensures the cache hit rate.

However, the number of packets in NIC-to-VM direction after re-aggregation may be too few. For example, we assume 30 packets are received from the NIC. After the re-aggregation ends, each VM has only 1-2 packets in the batch array, which greatly reduces the efficiency brought by batch processing. So in vSwitch, different batch sizes can be set for the NIC port and VM port respectively. This is also a common practice in the actual production environment, e.g. Google Cloud[69] sets the batch size of the NIC port to 120, and the batch size of the VM port is set to 30. Secondly, we provide a number threshold and a waiting time threshold for each packet in the batch array. So that after the re-aggregation operation, the PMD thread will check each VM's packets in the array, and process them only if the number of packets or the waiting time exceeds the threshold.

Algorithm 2 Batch processing of PMD thread 2

```
1:  function PMD MAIN LOOP
2:      while true do
3:          batch ← receive_from_NIC()
4:          PRECLS_processing(batch)
5:          for i = 0 → VM_cnt do
6:              rebatch[i] ← gather_with_destination(batch, i)
7:          end for
8:          for i = 0 → VM_cnt do
9:              update_tokens(C_avail[i])
10:             if rebatch[i].size == 0 or C_avail[i] < 0 then
11:                 continue
12:             end if
13:             start_tsc ← get_tsc()
14:             EMC_processing(rebatch[i], found_array)
15:             if rebatch[i].size! = 0 then
16:                 MFC_processing(rebatch[i], found_array)
17:                 if rebatch[i].size! = 0 then
18:                     send_upcall(rebatch[i], found_array)
19:                 end if
20:             end if
21:             execute_actions(found_array)
22:             send_to_VM(found_array)
23:             C_avail[i] − = (get_tsc() − start_tsc)
24:         end for
25:     end while
26: end function
```

4.3.2.2 Overhead analysis

According to the description in the previous chapter, the CPU consumption during the three stages of packet forwarding can be named as $C_{ingress}$, $C_{classification}$ and C_{egress}. Compared with the native OVS-DPDK, the packet matching in PRECLS is added to $C_{ingress}$. This part of the overhead is little and will not increase with the number of VMs. The overhead of re-aggregation will add to $C_{classification}$ and C_{egress}, and that is mainly due to the n (number of VMs) times of function calls at the beginning of each loop. So this part of the overhead will increase as the number of VMs increases. Fortunately, due to the large proportion of $C_{classification}$ and C_{egress} (more than 80%), the impact of n times of redundant logic code is trivial.

4.3.3 IO-dedicated CPU resources restriction

With all the shared flow tables are isolated, the attackers cannot interfere with other tenants' packet matching by "pulling" rules in the flow table. But they can still drop the experience of other tenants by competing for more IO-dedicated CPU resources[22, 21]. So we also need to isolate that. To achieve fine-grained CPU resources isolation, we provide each tenant with IO-dedicated CPU resources allocation at the "cycle" granularity based on the CPU-cycle based token bucket (C2TB) mechanism that we proposed in [47, 48]. Here we present briefly the two steps to achieve CPU resources isolation.

Modeling. In the first step, CSP can measure and model the relationship between CPU consumption and bandwidth on the specific software and hardware platforms. For example, under some bandwidth specifications (e.g. packet size in 64, 128, 256, 512, 1024 and 1518) and flow table status (e.g. number of entries or subtables), the correspondence can be established by generating traffic and measuring the consumption of IO-dedicated CPU resources in the vSwitch. And these relationships are stored in the configuration file. When a tenant purchases a VM, the required CPU resources (cycles/s) can be calculated according to the values in the pre-measured configuration file.

CPU resources limitation. The second step is to use the C2TB rate limiting method to restrict the IO-dedicated CPU resources for each VM. Different from the traditional token bucket algorithms that use the bits or number of packets as tokens, the tokens in C2TB represent the available IO-dedicated CPU cycles of each VM. The token generation rate of each VM is the CPU cycles/s we allocated to it. During each loop in PMD thread processing, after the PRECLS processing, we will update the tokens in each VM's token bucket (see line 9 in Algorithm 2). Then if the number of tokens is negative, we will jump this VM and process

packets for next VM. Only if the number of tokens is greater than 0, we will continue to perform the *classification* and *egress* for current VM. After that, the consumed CPU cycles will be subtracted from the token bucket (see line 24 in Algorithm 2).

4.4 Implementation

We implemented the D-TSE strategy based on OVS 2.9.2 with DPDK 17.11.2. The modifications are focused on the file *lib/dpif-netdev.c*, especially in the *pmd_thread_main()* function which is the main loop of PMD threads. In order to achieve FI, we modified the allocation principle of EMC and MFC, but did not modify their algorithms. In addition, we implemented an independent PRECLS classifier for each PMD thread. It is formed by a hash table based on the existing cuckoo hash function[97] in OVS. For the VM ports, the key in PRECLS is $\langle srcip \rangle$, while it is $\langle dstip, vni \rangle$ for the NIC port. When each VM instance is created and assigned an IP address, the correspondence of key and EMC address will be added to PRECLS. For realizing CPU resources isolation, we implemented a C2TB token bucket for each VM. It obtains the current CPU cycles through the *rdtsc* instruction[89], which only needs one cycle to read the value in register. So it can be widely used on the datapath without worrying about performance overhead.

All these modifications are easy to realize, and it takes about 200 lines of code to implement. We should note that the D-TSE is not only a strategy to prevent TSE attack, but also a design principle of vSwitches, that can be applied to more vSwitch designs and implementations in the future. We will further demonstrate in Section 4.5 that the performance cost of achieving isolation in vSwitch is negligible.

4.5 Evaluation

In this section, we first evaluate the defensive effect of D-TSE against TSE attack. Then we need to evaluate its cost – the performance degradation.

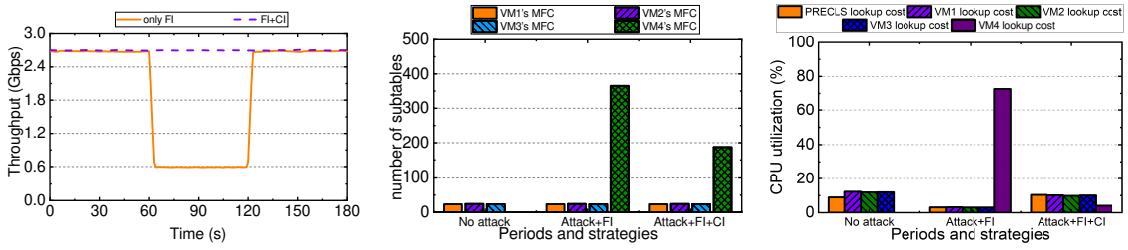
To build our cloud platform environment in these experiments, we use the following configurations: Intel Xeon CPU E5-4603 v2 2.20GHz (32 logical cores on 4 NUMA nodes), 64GB DDR3 memory at 1333MHz, one Intel 82599ES 10-Gigabit Dual Port NICs and Ubuntu 16.04.1 (kernel 4.8.0) as operation system. The cloud platform is built on QEMU 2.10, DPDK 17.11.2 and OVS 2.9.2. In OVS-DPDK, we use 2 PMD threads bound to 2 dedicated CPU cores to

forward packets in the VM-to-NIC and NIC-to-VM directions respectively (see Fig. 4.1). Each VM is assigned with 2GB memory and 1 logical core in all conditions.

4.5.1 Defensive effect

In order to measure the effectiveness of the proposed D-TSE mechanism, we measures the effect of TSE attack under Flow table isolation (FI) and CPU resources isolation (CI) successively. In this experiment, we run 3 victim VMs (the purchased bandwidth of VM1, VM2 and VM3 are all 3 Gbps), and use VM4 (purchased bandwidth of 1 Gbps) to generate malicious traffic within 60-120 seconds. The results are shown in the Fig. 4.6.

In Fig. 4.6(a), it can be seen that when only FI works, malicious tenants can still achieve attacks. This is because malicious tenants can still change the matching complexity in their own flow tables, seize more CPU resources, and then causes the CPU resource contention problem described in Chapter 3. In order to prove this point of view, we counts the number of subtables in each VM's MFC, and shows them in Fig. 4.6(b). We also calculates the CPU consumption for each VMs' packet processing, and draw the Fig. 4.6(c). It turns out that the FI can protect the victim VMs from malicious VM4 polluting the flow table entries and creating forwarding failure points. But the result of "Attack+FI" in Fig. 4.6(c) shows that VM4 can still occupy the CPU resources of the victim VMs. In this experiment, VM4 consumes ten times the CPU resources of the three victim VMs, which is also the main reason for the bandwidth drop of the victim VMs. Therefore, after integrating the CI provided by the C2TB mechanism, the TSE attack will no longer have any effect (see Fig. 4.6(a) "FI+CI" dotted line).



(a) victim VMs' throughput (b) The number of subtables (c) CPU consumption

Figure 4.6 The defensive effect under D-TSE strategy

In Fig. 4.6(c), we can also see the efficiency of PRECLS. The cost of searching in PRECLS with no attack is negligible and less than the cost of searching in flow tables of each VM, but it is almost the same when the attack is happening. That is not caused by the attack, but caused by more traffic from VM4 which needs to be processed in PRECLS. The most CPU-consuming

operation in PRECLS is to parse each field in the packet header, which is originally done in the EMC lookup. So if the batch size on the NIC port is increased, the CPU resources used by PRECLS searching will further decrease.

4.5.2 Multi-tenant performance

Besides isolation enhancement and attack defense, this section also evaluates the overhead of the D-TSE mechanism in terms of performance degradation. The overhead of D-TSE mainly comes from two aspects, the one time of hash lookup overhead in PRECLS, and the overhead in batch re-aggregation. According to the analysis in Section 4.3.2.2, since the overhead in batch re-aggregation is related to the number of VMs, it is necessary to measure the impact of the D-TSE mechanism on the VM network performance in multi-tenant scenario.

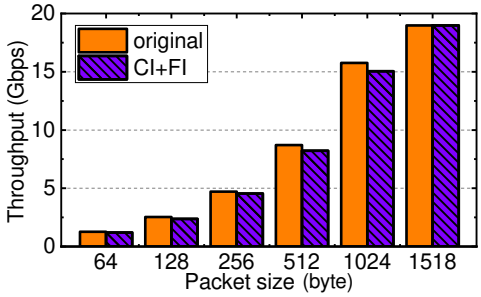


Figure 4.7 The performance comparison under multi-tenant scenarios

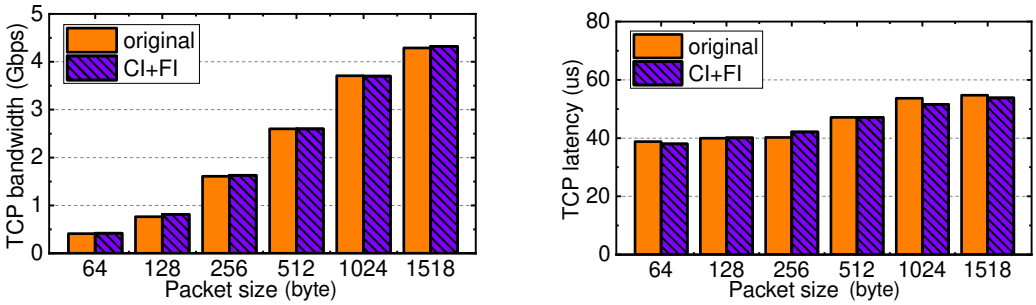
In order to measure the network throughput of VMs in multi-tenant scenario, we run 4 VMs on the same platform in this experiment. We use Spirent's TestCenter[96] as the traffic generator, and run DPDK l2fwd program in the VMs to forward traffic back to TestCenter. The results are shown in Fig. 4.7. Compared with the native OVS-DPDK ("original" in the figure), the D-TSE mechanism reduces the throughput of the vSwitch by 3–5%. The performance drop is the largest when sending 1024-byte packets , and the drop is smallest with 64-byte packets. From the experimental results, it can be seen that the D-TSE mechanism has little impact on the forwarding performance, and the difference in throughput drop is caused by the fact that cache hit rate has a greater impact on the forwarding performance of larger packets. In addition, in the case of forwarding 1518-byte packets, since the throughput exceeds the 20 Gbps link limit, the throughput under the native OVS-DPDK and D-TSE mechanisms are the same.

This experiment proves the effectiveness of batch re-aggregation, which prevents the network forwarding performance from being greatly reduced. In more extensive experiments, we find that as the number of VMs increased, the performance degradation caused by batch processing

breaking did not exceed 5% at most. If the CSPs are willing to increase the batch size on the NIC port, the throughput drop can be further reduced.

4.5.3 TCP performance

In addition to the network performance under multi-tenancy, we also consider the TCP performance inside a single VM, as this scenario will be directly related to the tenant's network experience.



(a) TCP bandwidth

(b) TCP latency

Figure 4.8 The TCP performance under D-TSE strategy

In this experiment, qperf[77] is used to measure the TCP performance within a single VM, and the results are shown in Fig. 4.8. Since the default kernel driver is used inside the VM, the main bottleneck for network performance is the kernel protocol stack processing. As can be seen from the figure, compare with the native OVS-DPDK, there is no significant difference in throughput or latency when using D-TSE mechanism. The difference between the two cases is within 1%. Therefore, although D-TSE slightly reduces the network forwarding capability provided by the vSwitch in the case of multi-tenancy, it has little impact on the experience of each tenant. That make it can be accepted by cloud platforms.

4.6 Conclusion

This chapter focused on the DoS attack and failure isolation issue in vSwitches of public cloud platform. Specifically, we prevented the TSE attack and a type of attacks, which exploit the isolation breakage that all tenants share data structure and physical resources for packet classification. To solve that, we proposed D-TSE strategy based on flow table structure isolation to isolate possible common failure point. In order to meet the challenges in design

and implementation, this chapter proposes a lightweight PRECLS to break the paradox in the design of separated flow table structure, and proposes an efficient batch re-aggregation mechanism to ensure performance. These innovations make the proposed mechanism easier to implement in the real environment. Finally, the evaluation on the OVS-DPDK platform proves that the D-TSE mechanism can isolate common failure point and prevented the data plane DoS attack represented by TSE, at the cost of less than 5% performance drop.

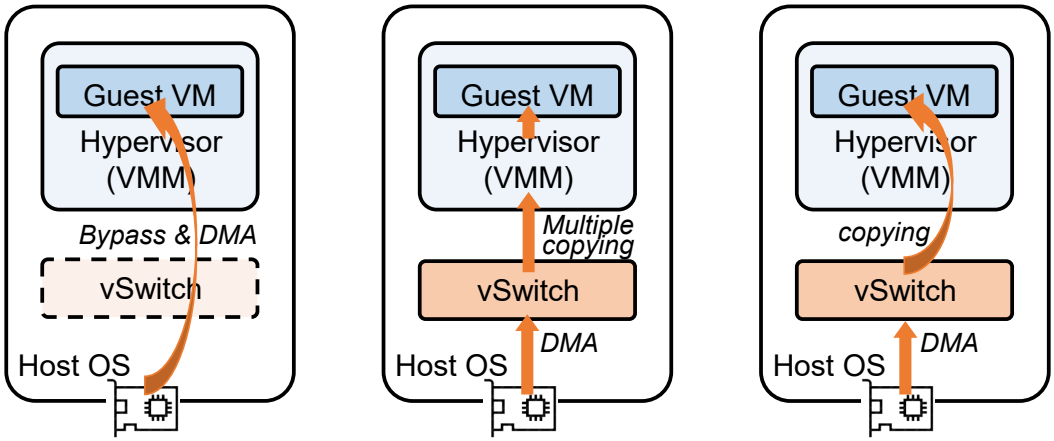
Chapter 5

S2H: Memory Access Isolation based Virtualized Network I/O Mechanism

5.1 Introduction

Cloud computing has become a popular paradigm for service provision, due to its ability to provide flexibility, dedicated execution and isolation to a vast number of services. These benefits are achieved thanks to advanced network virtualization techniques, which provide each tenant a VM with its own network topology and traffic control strategy [9]. The VM is an independent operating system running inside the hypervisor (also known as VMM) with an isolated running environment, and can flexibly reuse the resources on the physical server. To realize network virtualization, a software vSwitch is run to provide packet exchange and traffic control for these high-density deployed VMs. As its most crucial part, the Virtualized Network I/O (VNIO) technology permits the delivery of packets through different I/O paths connecting NICs to VMs.

There are mainly two types of VNIO solutions: hardware-assisted and software-based. Hardware-assisted solutions are shown in Fig. 5.1(a). They attain "bare-metal" performance by using Single Root I/O Virtualization (SR-IOV[98–100]) that can bypass the virtualization layer so that multiple VMs directly access a single NIC via different Virtual Functions (VFs). However, these solutions lose the flexibility enabled by virtualization (*e.g.*, memory overcommitment support[101], VM live migration[102], *etc.*) and face the risks of I/O channel attack[103]. For these reasons, hardware-assisted solutions are not widely adopted in enterprise cloud computing services [32, 104, 69, 105].



(a) hardware-assistance (b) full-virtualization (c) para-virtualization

Figure 5.1 The types of virtualized network I/O mechanisms

In contrast, software-based solutions can support full-virtualization (see Fig. 5.1(b)) and achieve a high-level of flexibility, but at the expensive cost of reducing system performance due to multiple times of packet copying. To alleviate this crucial issue, para-virtualization (see Fig. 5.1(c))is proposed for reducing the times of packet copying and more efficiently transferring the I/O data, with the sharing of memory between VMs and vSwitch. By providing a good trade-off between performance and flexibility, para-virtualized VNIO has been supported by all well-known vSwitches[106] and widely adopted in many real-world cloud platforms, *e.g.*, Google's Andromeda [69] and Alibaba Cloud[105] all adopt virtio based para-virtualized VNIO [71]. Nonetheless, the shared memory in para-virtualization goes against strict VM isolation and creates potential risks, *e.g.*, a malicious tenant may escape from its private VM environment and gain access to the shared memory that belongs to other VMs. This isolation issue significantly challenges the security and stability of cloud computing services.

In this chapter, we categorize existing para-virtualized VNIO solutions into two types of models, *i.e.*, VM to vSwitch (V2S) and vSwitch to VM (S2V), according to their memory-sharing models. In the V2S model, VMs share their private memory with a vSwitch process, that launches several PMD threads for the Packet Delivery (PD) tasks. As the user-space vSwitch process has the privilege to access all VMs' whole memory, the isolation between VMs and the host is broken. On the contrary, in the S2V model, vSwitch allocates a piece of monolithic I/O memory to share with all VMs and the PD procedure is completed inside each VM. A malicious VM may cross its boundary and access other ones' packets, which violates the isolation among VMs.

These isolation issues brought by insecure memory-sharing models have already been noticed, but the reinforcement solutions are all at the expense of significant degradation in performance. For example, the community has reported a type of DMA attack under the V2S model[39, 41, 40], and they proposed a solution called vIOMMU[41–43] to restrict the memory access by reinforcing the address translation procedures. Unfortunately, this solution can only prevent illegal memory access during PD operations, and the insecure shared memory still exists. To make matters worse, the system performance after using vIOMMU will be severely degraded to $\sim$20%. Other memory access protection mechanisms, such as the hardware-based SGX[89], are rarely used in the data path of the systems and the use case in Network Function Virtualization (NFV) scenarios has shown that these hardware-based protection mechanisms also severely reduce performance[45].

To effectively guarantee VM isolation under the premise of ensuring performance, we propose a new memory-sharing model for para-virtualized VNIO called S2H, which exploits the hypervisor to transfer I/O data between VMs and vSwitch. Compared with the S2V and V2S models where vSwitch and VM communicate directly, S2H uses the hypervisor as an intermediate "setter" [1] that isolates VM and vSwitch. Since the hypervisor is maintained by the service provider and has access to the VM memory by default, it is more reliable to use it for memory sharing and packet delivery. On the aspect of performance, in order to maintain the advantage of high throughput brought by memory-sharing, some more innovations in concurrent memory access and scalability are required. First, an efficient framework for memory sharing and access is needed to support the transfer of packets between vSwitch and a number of hypervisor processes. Second, we need to efficiently schedule the concurrent PD procedures, which are distributed in hypervisor processes, to improve the VNIO scalability.

Thus the main contributions of this chapter are summarized as follows:

- We classify the memory-sharing models of existing para-virtualized VNIO solutions into S2V and V2S, and then analyze how they violate the memory isolation. To guarantee the isolation without degrading the system performance, we propose a new S2H model that adopts hypervisor processes for sharing memory with vSwitch and completing PD procedures.
- To efficiently deliver packets via the shared host I/O memory between the vSwitch and hypervisor processes, we take advantage of the DPDK memory management in

[1] The "setter" in volleyball sport is responsible for passing the ball, and we use it here to describe that hypervisor delivers I/O data between VM and vSwitch.

hypervisor processes for concurrent memory access, and well design the conflict-free packet delivery pipeline for high-speed packet processing.

- To support scalability and run a large number of VMs on a single server, we propose a "batch-grained" scheduling strategy, which schedules PD procedures on limited CPU cores according to batch processing workload instead of uncontrollable time slices. This kind of scheduling strategy brings more efficiency and flexibility.

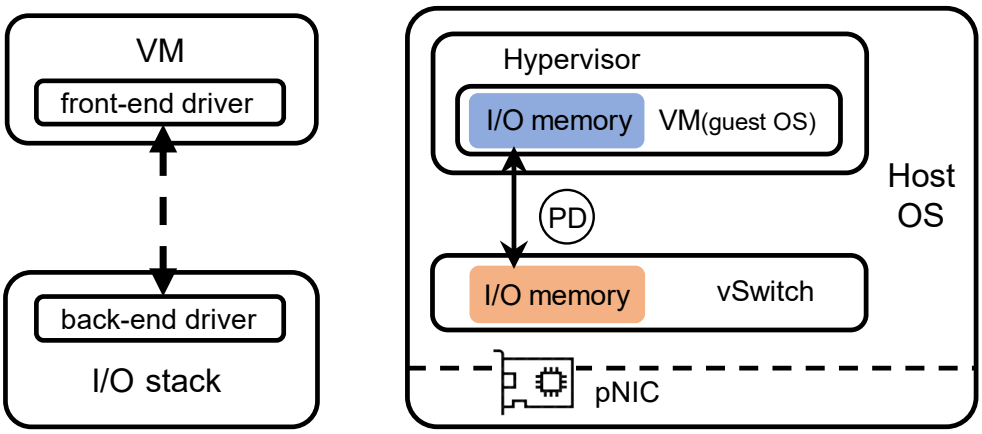
- We implement the S2H prototype based on the vHost-User architecture as it is the *de-facto* para-virtualization standard which has been widely adopted in commercial products. We show that S2H can be simply realized by adding less than 1000 lines of code to the existing solutions. Extensive evaluations are conducted in diversified scenarios and settings including data-path performance, inter-VM performance, application performance, and scalability. The results show that S2H can achieve comparable throughput with 9% more latency than those of the native vHost-User architecture, while effectively guaranteeing VM isolation.

5.2 Problem define and motivation

In virtualization technology, the software that creates and runs VMs is called VMM or hypervisor. The host is the physical server on which a hypervisor runs one or more VMs. Each VM is called a guest VM. Generally, a hypervisor contains both user-space processes and kernel modules. For example, in the QEMU/KVM[5, 6] implementation, a hypervisor layer consists of independent user-space QEMU processes and a KVM module. Each VM is actually virtual CPU (vCPU) thread(s) launched by the corresponding QEMU process and interacts with the KVM module.

VNIO is an important building block of virtualization, and it is responsible for delivering traffic among VMs and NICs. In a full-virtualization VNIO solution, the hypervisor serves as a dividing layer between the host and the guest VM. The hypervisor emulates a full function of NIC, that can be driven by the native driver in a guest VM. The full-virtualization is flexible, but the device emulation causes significant performance overheads [107]. Authors in [108] proved that optimizing software interrupts for reducing VM-exit in device emulations can greatly improve the performance of full-virtualization. So in order to break the bottleneck in device emulations and software interrupts for better I/O performance, various para-virtualization solutions are proposed [71].

As shown in Fig. 5.2(a), a para-virtualized VNIO solution consists of two parts: a front-end driver in the guest VM and a back-end driver in the host. The front-end handles the virtual device emulation in the guest Operating System (OS), while the back-end performs I/O data transfer between host and guest VM. In this chapter, we focus on the back-end as the I/O operations mainly happen there.



(a) architecture (b) data path

Figure 5.2 The architecture and data path of para-virtualized network I/O

Fig. 5.2(b) illustrates a typical para-virtualized VNIO environment, with one VM running on the host. As it shows, the back-end consists of a PD procedure and vSwitch. The PD procedure exchanges I/O data between the two blocks of I/O memory residing in the VM and the vSwitch, respectively, while vSwitch is responsible for forwarding traffic among virtual and physical ports. For example, if a packet arrives at a port in the NIC, the vSwitch component first captures and holds the packet in the I/O memory (the orange colored block in Fig. 5.2(b)). The vSwitch then parses the packet, looks for its forwarding information, and finally notifies the PD procedure to copy the packet from the host to a particular VM.

It is worth noting that, these two blocks of I/O memory in Fig. 5.2(b) belong to different memory spaces, i.e., host memory space and VM memory space[2]. When the back-end (vSwitch) used to run in the kernel-space, it can access VM memory and do packet copying by default. But in the last decade, to improve performance, vSwitch has been transferred from the kernel-space (known as the bridge in Linux kernel) to user-space to exploit high-performance drivers like DPDK and netmap [57]. As a result, the vSwitch and the VM are isolated from each other's memory accesses. To implement the memory copying operations in the PD procedure, a

[2]The host memory space here indicates the memory space of vSwitch process. As the vSwitch is deployed directly on the host OS and we need to distinguish its memory from VM's memory, we introduce the concept of host memory space.

memory-sharing model is needed, i.e., either sharing the I/O memory from vSwitch to VM (S2V), or from VM to vSwitch (V2S), which introduces isolation issues.

5.2.1 Existing memory sharing models and isolation issues

The shared memory largely boosts the I/O performance, as the PD procedure can directly access both the source and destination memory addresses during memory copying. However, there is a trade-off between performance and isolation. As the motivation of this work, we classify and analyze the user-space memory-sharing models in para-virtualized VNIO solutions.

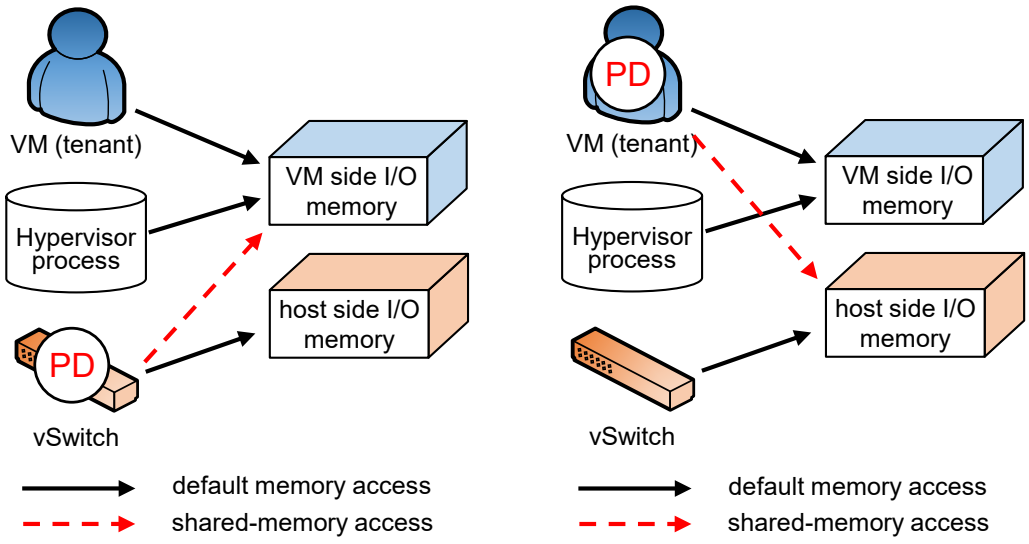
According to Fig. 5.2, the memory-sharing model consists of two blocks of I/O memory (on VM and host sides), three participants (VM, hypervisor, and vSwitch), and a PD procedure. As the PD procedure needs the privilege to access the I/O memory on both sides, the choice of I/O memory-sharing models depends on the location where the PD procedure performs, and vice versa.

Depending on the locations of the shared memory and PD procedure, we categorize the existing para-virtualization solutions into 2 models, S2V shown in Fig. 5.3(a) and V2S shown in Fig. 5.3(b). In each figure, the PD procedure and three participants (VM, hypervisor, and vSwitch) are on the left, while the two blocks of I/O memory are on the right. The arrow lines present the memory access from the participants to the I/O memory. In particular, a solid line indicates that the memory access is granted by default, while a dashed line indicates that the access is implemented via the memory-sharing model.

Fig. 5.3(a) illustrates the **V2S model**, which is followed by virtio vHost-User[31], Google's Andromeda[69] and ELVIS[32]. Taking the virtio vHost-User solution in QEMU/KVM implementation as an example, VM and QEMU hypervisor are granted access to the VM's memory, while the vSwitch is granted the access to the I/O memory on both sides. For vSwitch, the access to the host side I/O memory is granted by default, and the one to the VM side I/O memory is implemented by the V2S memory-sharing model via a series of *mmap()* operations. Several threads (one in default, more with the increase in the VM count or workload) are created in the vSwitch process to deliver packets for all VMs.

Fig. 5.3(b) illustrates the **S2V model**, which is followed by NetVM [34], IVSHMEM [35, 36] and ClickOS[73]. Compared with the V2S model in Fig. 5.3(a), VM is granted access to the block of I/O memory on the host side. Meanwhile, the PD procedure is moved from the host side to the VM side. Taking the NetVM VNIO solution as an example, vSwitch allocates a block of memory and shares it with the VM. A virtual PCI (vPCI) device is created in the VM,

and the device memory is redirected to the block of shared-memory. Therefore, each VM can complete its PD procedure via the virtual device.



(a) The V2S model (b) The S2V model

Figure 5.3 The existing memory-sharing models

The Table 5.1 lists all current VNIO mechanisms and which memory-sharing models they belong to. It can be seen that almost all VNIO mechanisms use the S2V or V2S memory sharing model. But unfortunately, both the S2V model and the V2S model commonly have the isolation issue either among VMs or between the VM and the host. The issue is critical in the cloud computing environment, where the per-host VM density is sufficiently high[109]. Taking the V2S virtio vHost-User as an example, the shared memory should be ideally restricted within the I/O memory region in the VM. However, we cannot predict the I/O memory addresses if they are dynamically allocated at run-time. As a result, the VM's whole memory is shared with vSwitch in the practical usage of virtio. The user-space vSwitch process, granted access to all VMs' whole memory, can easily become the Achilles' heel. Further, existing vSwitch implementations are not reliable. Even OVS[59, 60], the most popular vSwitch project, has security vulnerabilities reported in the Common Vulnerabilities and Exposures (CVE) database[110, 111]. With a compromised vSwitch process, the hijacker can arbitrarily access and even overwrite any piece of memory in VMs via I/O operations. The community has already noticed the security issue, and a type of DMA attack has been reported[39, 41, 40]. The issue also exists in NetVM and IVSHMEM S2V models. Malicious tenants inside the VM could rewrite the vPCI device driver or exploit existing vulnerabilities and software bugs in the driver to access the host I/O memory without any restrictions[112].

Table 5.1 The comparison of existing virtualized network I/O mechanisms

Solution/Mechanism	Memory-sharing model	Performance	Isolation
vHost-User[31]	V2S	●	○
Andromeda[69]	V2S	●	○
ELVIS[32]	V2S	●	○
Xen1[33]	V2S	●	○
NetVM[34]	S2V	●	○
IVSHMEM[35, 36]	S2V	●	○
clickOS[73]	S2V	●	○
Xen2[37]	S2V	●	○
Xen3[38]	V2S	○	●
vIOMMU[41, 42]	V2S	○	●
Zcopy-vhost[113]	N/A (page flipping)	○	●
Hyper-switch[114]	N/A (kernel-space PD)	○	●

To solve these isolation problems, the current related works are devoted to simply restricting the memory access under existing architectures. For example, to reinforce memory address translation functions, vIOMMU was proposed to prevent DMA attacks[41–43]. Unfortunately, these reinforcement attempts have severely degraded the VNIO performance (to $\sim$20% of the original performance) and therefore cannot be adopted. Other memory access protection mechanisms, such as the hardware-based SGX[89], are rarely used in the data path of the systems and the practice in NFV scenarios has shown that these hardware-based protection mechanisms also severely reduce performance[45].

We also notice that some other VNIO mechanisms take a different approach. They do not use shared memory, but seek kernel-level EPT[115] page table mapping, or directly implement data path in kernel space[113, 114]. But they come at a price of low performance. It has been proved that the performance of today's kernel-space packet processing under the acceleration of the latest XDP[116] and eBPF[117] technologies is still far less than the datapath in user space[118], so it is not necessary to roll back VNIO performance to ten years ago for isolation.

In addition to the isolation design for VNIO, this chapter also investigates some other memory access protection mechanisms, such as hardware-based Intel SGX[89] and ARM TrustZone[119]. However, the basic principle of this type of technology is to encrypt key memory. Even the OS cannot directly obtain data from memory without authorization. Although it can provide the best memory security, it will cause large encryption and decryption overhead for memory-intensive applications. The practice in the NFV scenario shows that the system performance can be reduced by 90% just by placing the flow table rules in the protected memory[45].

5.2.2 Motivation

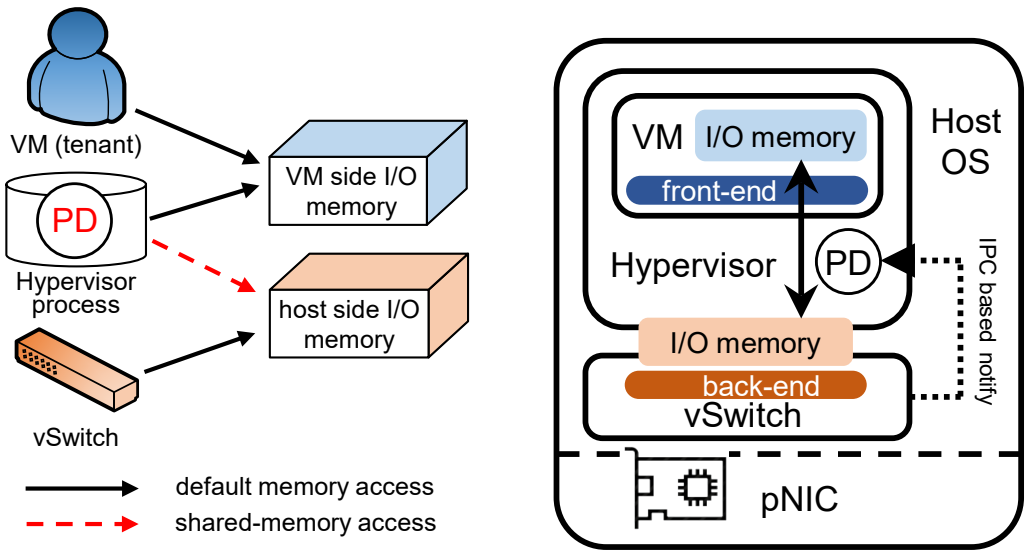
Rethinking the two models shown in Fig. 5.3(a) and Fig. 5.3(b), we can conclude that because packets need to be delivered between vSwitch and VM, it is natural to make them share the memory with each other. But both models have isolation issues and cannot be simply reinforced. It is noteworthy that, until now the hypervisor has not participated in either memory-sharing or PD procedure. However, if the hypervisor process can complete the PD procedure, it has two advantages. First, the hypervisor process can access VM's whole memory by default without memory-sharing. Taking the QEMU/KVM virtualization as an example, each time when booting a VM, a QEMU process needs to be launched with command line parameters that present the properties of a VM. These VM properties include the memory size, the devices' information, *etc*. The QEMU process then emulates devices and allocates the memory for the VM according to these parameters. Therefore, each QEMU can access the memory of its corresponding VM. The second benefit is, compared with the uncontrollable behavior inside the individual VMs, compromising a hypervisor process is known to be much more difficult as it is typically well maintained and monitored by the platform provider[120]. This motivates us to re-examine the existing strategies, and design a new memory-sharing model and architecture with the use of hypervisor to facilitate the memory access.

5.3 Secure memory sharing model design

According to the motivation of using hypervisor to achieve secure memory sharing, in this section we will describe the proposed memory sharing model in detail, and make a comprehensive isolation and security analysis, then lead to the implementation challenges.

5.3.1 S2H model

We propose a new memory-sharing model (see Fig. 5.4(a)) which shares the host-side I/O memory from vSwitch to Hypervisor (S2H) process. Significantly different from S2V and V2S models, neither VM nor vSwitch process in the S2H model can access the I/O memory that does not belong to itself. Instead, the hypervisor is granted to access the blocks of I/O memory on both sides. Consequently, the PD procedure is moved to the hypervisor layer, and a hypervisor process launches a dedicated PD thread for the corresponding VM.



(a) S2H model (b) S2H VNIO architecture

Figure 5.4 The secure memory-sharing model and virtualized network I/O architecture

Fig. 5.4(b) shows the S2H VNIO architecture, where the hypervisor containing a PD procedure works as a "setter" between the front-end and the original back-end. If the PD procedure follows the existing communication protocol, neither front-end driver nor back-end (the part in vSwitch process) needs any modification.

5.3.2 Secure isolation analysis

The main goal of this work is to resolve the VM isolation issues to improve the security of para-virtualized VNIO. We first analyze the isolation capability of the proposed S2H model from two perspectives, isolation among VMs and isolation between a VM and the host.

Isolation among VMs: In the S2V model, a malicious tenant could exploit existing vulnerabilities and software bugs in the vNIC driver to acquire or modify other tenants' I/O data via out-of-boundary memory access on the host side. In the proposed S2H model, no I/O memory is directly shared with the VM. Instead, the PD procedure in the hypervisor works as a barrier to prevent unauthorized memory access. As a result, S2H has better isolation among VMs.

Isolation between VM and host: In the V2S model, taking the virtio vHost-User as an example, the host-side user-space vSwitch process is granted access to all VMs' whole memory. If a vSwitch process is compromised, a hijacker can arbitrarily access and even overwrite any

piece of memory in VMs via I/O operations. In contrast, in the proposed S2H model, no I/O memory is shared from VM to the host, so it has better isolation between a VM and the host.

Table 5.2 The memory access comparison under different memory-sharing models

model	each VM	each hypervisor	vSwitch
V2S	VM	VM	hostbuf+all VMs
S2V	VM+hostbuf	VM+hostbuf	hostbuf
S2H	VM	VM+hostbuf	hostbuf

To illustrate the improved security owing to better memory isolation, we show the memory range that each component can access under different memory-sharing models in Table 5.2. In the table, we can see the proposed S2H minimizes the shared memory size. Compared with the S2V and V2S models, S2H improves the security in two aspects. On the one hand, according to existing security issues, the most common DMA attacks[39] and buffer overflow attacks [121, 111] on VNIO require direct manipulation of sensitive memory addresses. As the proposed S2H can isolate the memory spaces of VM and vSwitch, these well-known attack models cannot work. On the other hand, the S2H introduces few security risks while providing better isolation. The hypervisor is the foundation of the virtualization environment and is commonly well maintained by service providers. Compared with VM operating systems or vSwitch software, the hypervisor is more secure since its size is relatively small and the exported attack surfaces for guest domains are considerably less[120]. Last but not least, in the proposed S2H model, we only add the PD procedure (memory copying workload) to the hypervisor. That means the introduced code is very low and executes as a separate thread which is independent of the existing code in hypervisor. So the newly added PD procedure is easy to control and maintain.

5.3.3 Implementation challenges

Despite the potential of providing better isolation and security, the realization of the proposed S2H model faces two major challenges. First, as a “setter” process between the front-end driver and the back-end component (vSwitch process), the PD procedure in the hypervisor adds the overhead to the VNIO processing. The sharing of memory between the vSwitch process and the concurrent hypervisor processes is more complex and will affect the performance. Secondly, the CPU resources that can be occupied by PD are very limited. Cloud service providers prefer to assign most of the computing resources to VMs and leave very few for VNIO processing. In the Google cloud, no more than two physical CPU cores per physical server are assigned to the

PD procedures [69]. Therefore, efficiency and scalability are the main design challenges to realizing S2H. We will enhance the efficiency of S2H from two perspectives.

- **Sharing memory among concurrent processes.** As each VM has a dedicated PD procedure in its parent hypervisor process, the shared memory will be accessed by multiple threads that belong to different processes. The block of shared memory is divided into small pieces to store I/O data and notifications belonging to different VMs, which are occupied, modified and freed by the concurrent PD threads and vSwitch processes. We need an efficient memory sharing framework to deal with the potential conflicts and contention from concurrent processes.
- **Thread scheduling.** As PD procedures run concurrently on limited CPU resources, scheduling these threads with a granularity of time slice is inefficient and will increase the competition, as well as context switching[83]. We need an efficient scheduling mechanism with granularity that can be properly set according to the working mode of PD procedures.

5.4 VNIO mechanism design and prototyping

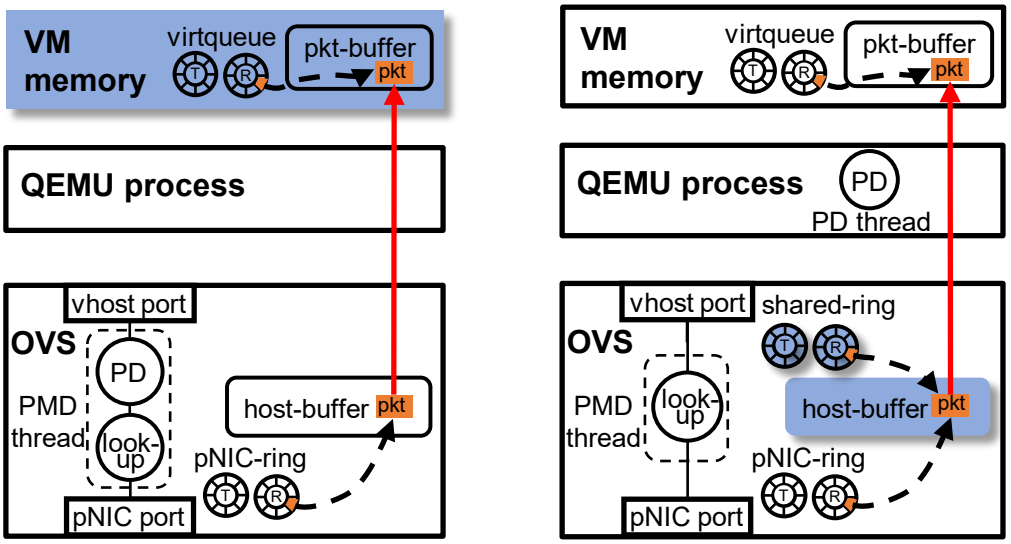
To address the above challenges of realizing S2H, in this section, we will elaborate on our proposed innovations for efficient memory sharing among concurrent processes and scalable scheduling of PD threads. We introduce our designs in the context of a prototype system. However, our sharing and scheduling strategies are general and can be used to guide the design and implementation over other VNIO architectures.

To demonstrate the function and prove its simplicity in realization, we prototype the S2H system over the vHost-User (V2S) architecture, as it is a state-of-the-art design and has been widely used in the production environment. The virtio *de-facto* standard that vHost-User follows is also supported by the QEMU/KVM virtualization platform, as well as the kernels of most operating systems such as Linux and Windows.

In this section, we first introduce the basic vHost-User platform and our prototype implementation, and then describe the principles and realization of our proposed memory sharing and thread scheduling mechanism.

5.4.1 vHost-User and basic platform

On the vHost-User architecture, the user-space QEMU process and the KVM kernel module provide VMs with the virtual execution environment. A corresponding QEMU process is created for each VM. OVS[59, 60, 122], as the back-end in vHost-User, handles the packet classification and forwarding. For high-speed packet processing, OVS is compiled into the DPDK framework[11], which possesses unique features, such as user-space network driver and efficient memory management. When the OVS-DPDK is running, it launches one PMD thread by default, to look up packet destination in flow tables and implement the PD tasks on all physical and virtual ports. The number of PMD threads in OVS-DPDK can be increased as the workload goes up, and each PMD thread must be bound to one dedicated CPU core.



(a) vHost-User implementation (b) S2H prototype implementation

Figure 5.5 The implementation of vHost-User and S2H prototype

The data exchange of vHost-User architecture is shown in Fig. 5.5(a). As a typical V2S model, the QEMU process does not participate in either the memory-sharing or the PD procedure. On the contrary, OVS is granted access to the memory both in the host and in VM. According to the function, a PD procedure can be divided into two parts: the notification path for the transmission of packet descriptors and the data path for packet transmission. There are also two types of memory infrastructure involved. In the first type, rings are used to buffer packet descriptors, e.g. the pNIC-ring stores packet descriptors in the host-side memory and the *virtqueue*, as part of virtio standard, stores descriptors on the VM side. As the second type, packet buffers are applied to store packets on both the host side and the VM side. The host-buffer is a block of memory managed by DPDK and used for storing packets received from all OVS's ports,

while pkt-buffer is allocated by the VM driver and used for receiving packets from hosts (e.g. skb_buff in Linux kernel).

Both notification path and data path are bidirectional. We take the traffic-in-VM as an example. After NIC receives packets and stores them in the host-buffer with DMA, the OVS PMD thread can poll the pNIC-rings to access the packets according to the descriptors. It decides which VM is the destination based on the matching of the parsed packet header with the entries in the flow table. Then, the PMD thread finds the available pkt-buffer addresses from *virtqueue* of the destination VM and copies packets from the host-buffer to the pkt-buffer. After the packet copying is completed, the PMD thread updates the *virtqueue* of the receiving direction to notify VM for getting packets. The procedure in the opposite direction is similar.

5.4.2 Prototype design and implementation

We modify the vHost-User architecture and implement S2H as shown in Fig. 5.5(b). The key of S2H architecture is that each QEMU process acts as a "setter" and runs a PD thread between the corresponding VM and OVS to provide the VM isolation. Since each VM has its own QEMU process to complete the PD procedure, OVS needs to share the memory with all QEMU processes on this server. To keep the data path unchanged and avoid additional overhead, we use the host-buffer as the shared memory, and OVS directly exposes the host-buffer to all QEMU processes. On the notification path, due to the need of transmitting packet descriptors between QEMU and OVS, a pair of separated shared-rings is allocated and shared between each QEMU and OVS process. Each time when OVS needs to send packets to VM, its PMD thread only needs to copy the packet descriptors from pNIC-ring to the particular shared-rings to be accessed by the corresponding QEMU process, according to the flow table lookup results.

To efficiently complete tasks on both data path and notification path, we create a dedicated PD thread running in the polling mode for each QEMU. The major workload of the PMD thread on each vhost-port in vHost-User solution is transferred to the PD thread of the corresponding QEMU process. As a result, the data path workload remains about the same as that there is no extra packet copying added, comparing with the primitive solution in Fig. 5.5(a). Meanwhile, as shown in Fig. 5.5(b), only one extra packet descriptor copying operation is added to the PMD thread of OVS on the notification path. As the descriptor data structure only contains the packet address information, the increased overhead is negligible. Now we also take traffic-in-VM as an example. After NIC receives packets and stores them in the host-buffer via DMA, the OVS PMD thread can poll the pNIC-rings to get the descriptors for the access of packets in the host buffer. Then after the flow table lookup, it copies the packet descriptors from pNIC-rings to

the destination QEMU's shared-rings and the PMD thread completes its job. In QEMU, the PD thread polls shared-rings and obtains packet descriptors pointing to the host-buffer. The following steps are the same as those of in vHost-User—getting available pkt-buffers, copying packets and updating the *virtqueue*.

Compared to the vHost-User, the heaviest workload, packet copying in the PD procedures, is undertaken by the PD threads in QEMU processes. The work done by the PMD threads in OVS is reduced to only the lookup in the flow table and the copying of packet descriptors. As the workload changes, we should allocate most of the CPU resources used by the OVS PMD threads in the native vHost-User to the PD threads of QEMUs. We also design a thread scheduling mechanism for these threads to run on limited CPU resources, which will be described in Section 5.4.4.

The native vHost-User is modified to prototype the proposed S2H. We only add less than 1000 lines of codes into QEMU to implement the PD thread and make a few changes to OVS. As the interface to the *virtqueue* from the PD thread remains unchanged from that of vHost-User, the components inside VM (such as the kernel, VM driver and *virtqueue*) do not need to be modified and are compatible with the configurations in native vHost-User. Though the prototype implementation is based on QEMU/KVM virtualization platform, S2H can also be adopted to other virtualization platforms (e.g. Xen, VMware), as a general VNIO architecture.

5.4.3 Memory sharing with concurrent access

As the foundation of VNIO, the shared memory is used to exchange packets between VMs and the host. For example, in vHost-User, each VM shares its memory space (blue in Fig. 5.5(a)) to the OVS process. It is very efficient and there are no conflicts during the shared memory access as it is essentially a single-producer single-consumer (SPSC) issue. But in S2H, the shared memory (blue in Fig. 5.5(b)) consists of two parts: the public host-buffer part and the private shared-ring part. The public shared host-buffer is a block of memory, that will be operated (allocate, access and free) by the PD threads of different QEMU processes and the OVS PMD thread simultaneously. So it will lead to a multi-producer multi-consumer (MPMC) issue. For each pair of shared-rings, it is newly added and only shared privately between the corresponding QEMU process and the OVS process.

We first introduce how we build these two parts of shared memory. For the public shared host buffer, as the fact that *mbuf pool* is managed by OVS-DPDK in the native vHost-User, we directly let OVS-DPDK share the *mbuf pool* with all QEMU processes as shown in Fig. 5.6.

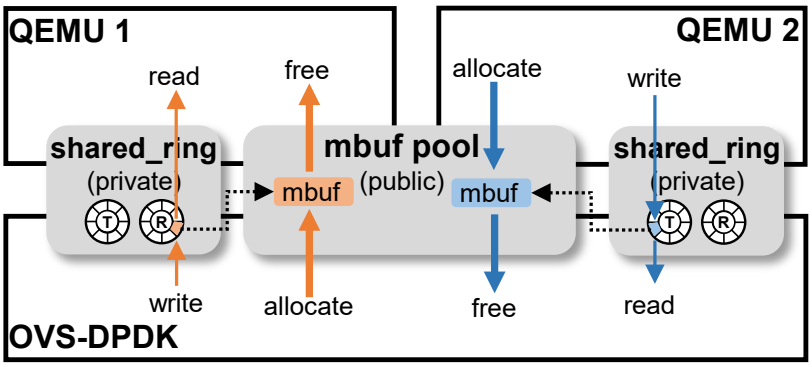


Figure 5.6 The concurrent shared memory access in S2H

To let these QEMU processes have access to the *mbuf pool*, we add "-mem-file = /path-to-rte_config" to the command line parameters for each QEMU process to find and map the *mbuf pool* to its own address space. Once a QEMU process starts and the initialization is completed, the *mbuf pool* has been accessible by its PD thread. For shared-ring, OVS will create a separate small piece of memory for each QEMU process. Then, each QEMU process attaches to this block of private shared memory. Both types of shared memory are constructed by calling *mmap()* functions to map the files into the process memory space.

To solve the MPMC issue in the public shared host buffer, we take the advantage of DPDK memory management to make the PD threads of all QEMU processes access this block of memory efficiently. As the access to the shared *mbuf pool* is in the form of reading or writing packets, we need unified memory management and also an efficient conflict-free packet processing pipeline. As the DPDK memory management is designed for processing packets at high speed to efficiently solve the MPMC issue, we implement the same data structures in QEMU as those in DPDK for accessing the *mbuf pool*. With the same memory management, both QEMU and OVS-DPDK can allocate or free *mbuf* (*mbuf* is a data structure used for storing a single packet, like skb_buffer in Linux kernel). So we design a conflict-free packet delivery pipeline as shown in Fig. 5.6. The allocation and free of *mbuf* are performed by the most suitable process according to the direction of the PD procedure. For example, the packet sender process (whether it is OVS-DPDK or QEMU) is responsible for allocating *mbuf* from *mbuf pool*, and the receiver process needs to free it after fetching the packet. Combined with the efficient DPDK memory management, packets are able to be transferred among processes via *mbufs* at high speed.

In the two kinds of shared memory, host-buffer (*mbuf pool*) is consistent with that in vHost-User, but the private shared-ring is newly added. As an intermediate, temporary storage area for the transfer of descriptor, its size may affect the S2H VNIO performance. We construct

performance evaluation with varying traffic loads, and find that when the shared-ring size is 4 times greater than the batch size, the system can completely deal with traffic bursts and achieve a reasonable performance. However, if the ring size continues to grow, the throughput will not rise anymore. Thus, we set the shared-ring size to be 4 times of the batch size.

Besides the performance concerns, the shared memory in S2H also needs to support other features in VNIO like reconnection and live migration, which are crucial in a practical environment. We propose the following schemes for realizing these features under our shared memory implementation. For reconnection, according to the order of shared memory initialization procedure, we let QEMU processes do the *munmap* and free operations on the *mbuf pool* immediately after OVS goes down. When the OVS restarts and successfully allocates the new *mbuf pool*, then all QEMU processes attach to it again. For the live migration, because the shared memory is allocated by OVS, a VM cannot be migrated to another server unless all the packets in the shared memory are processed. As a result, in S2H, before QEMU starts the migration, we need to ensure that all packets referred in the shared-ring have been delivered to VM.

5.4.4 Scalability Support

Scalability is an important feature of the cloud platform, which contains two aspects: scalability among VMs and scalability inside a VM. For cloud service providers, the cost is one of their biggest concerns. In a multi-tenant scenario, the biggest scalability issue among VMs is how to use limited CPU resources to support the PD procedures of a large number of VMs. For a single VM, how to increase the receiving and sending queues in vNIC is the key to improving the VM network performance and achieving the scalability inside the VM.

In native vHost-User, OVS PMD threads naturally support these two types of scalability. As mentioned before in Section 5.3, centralized PMD threads poll all the *virtqueues* of each VMs in sequence and do PD on the corresponding ports. The number of PMD threads is equal to the number of CPU cores that are required to complete PD procedures. For multi-queue, OVS needs to add multiple *virtqueues* of a single VM to the polling queue of the PMD threads. It works well for the two types of scalability, and the only limitation is that the PDs of all VMs are executed sequentially and are not flexible enough.

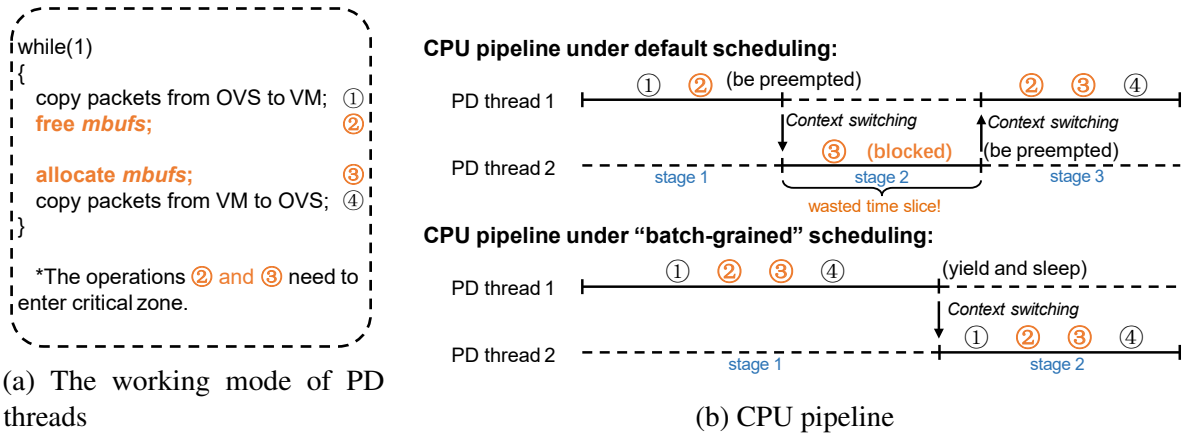
In S2H, the situation is the opposite. As the PD procedures are distributed in PD threads of different QEMU processes, it brings the nature of flexibility. The out-of-order execution of different VMs' PD procedures can support some SLA and QoS strategies of service providers

that were not available in traditional architectures. For multi-queue, because the interaction between the PD thread and *virtqueue* remains the same, we only need to increase the number of shared-rings to the same number as *virtqueues*. But as each VM has a dedicated PD thread to perform its PD procedure, the number of PD threads would be relatively large. We need to bind multiple PD threads on limited CPU cores and schedule the PD threads on the same core to achieve scalability among VMs.

5.4.4.1 "Batch-grained" thread scheduling

However, binding many PD threads on the same CPU core will bring serious performance issues. By default, these PD threads bound to the same CPU core are managed under the Linux scheduling policy "SCHED_OTHER", which is a kind of completely fair scheduling (CFS) policy. Under the CFS policy, each PD thread is allocated with a fixed time slice to run its workload. After the time slice is used up, the core will be preempted by other PD threads. As the scheduling granularity is too small, frequent context switching among different PD threads will result in huge performance overhead. In addition, PD threads may be preempted when operating the shared data structure and then lock each other. Usually, this situation happens when the PD thread enters the critical zone. For example, as shown in Fig. 5.7(a), allocating and freeing *mbuf* requires the threads to enter the critical zone, which is common in the main loop of PD threads. We use the CPU task pipeline under default scheduling in Fig. 5.7(b) to show this issue. Each "stage" in this figure represents the time period of a PD thread occupying the CPU to execute workload until the context switching. In stage 1, the PD thread 1 is still in the critical zone when being preempted by PD thread 2. In stage 2, PD thread 2 will be blocked when it tries to enter the critical zone to allocate *mbufs*. As a result, the time slice is wasted in stage 2. In stage 3, PD thread 1 completes its work and leaves the critical zone, then PD thread 2 will be able to continue its workload after it preempts CPU in the next stage.

We propose a "batch-grained" scheduling strategy to avoid blocking and make it more CPU-friendly. For efficiency, the scheduling granularity needs to be well designed: how long each PD thread can occupy the CPU and when it can yield the CPU. We use one batch as granularity. First, we set all PD threads' scheduling policy to "SCHED_FIFO". Because the threads under "SCHED_FIFO" will not be preempted unless they yield the CPU by themselves. That gives us the ability to determine the granularity of the schedule by workload rather than uncertain time slice. As the main loop of each PD thread contains polling *virtqueue* and shared-ring to transfer one batch of packets, we allow each QEMU's PD thread to run one loop before it goes to sleep and yields the CPU for other threads. The pipeline is shown in Fig. 5.7(b), it can be



(a) The working mode of PD threads

(b) CPU pipeline

Figure 5.7 The "batch-grained" scheduling

seen this kind of scheduling strategy avoids the blocking caused by context switching in the critical zones and also makes the context switching less frequent.

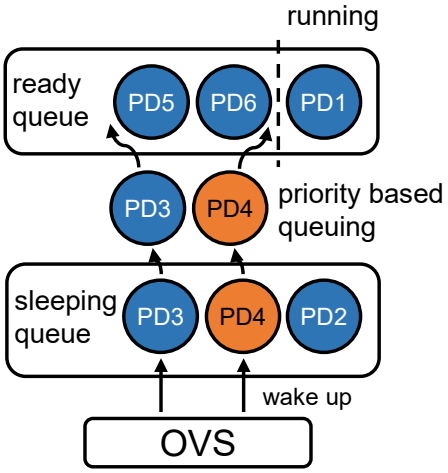


Figure 5.8 An example of SLA strategy under "batch-grained" scheduling

Besides efficiency, the "batch-grained" scheduling also provides more flexibility that can be used to design more complex SLA policies. Its details can be found in Section 3.4[47, 48]. Here we show it can be easily implemented in S2H without modifying the architecture. The logic is shown in Fig. 5.8, each PD thread is set with a priority and sleep time threshold. OVS wakes up the corresponding PD thread based on the sleep time and whether there are enough packets to send. But the woken-up PD threads do not immediately occupy the CPU, they stay in a ready queue based on arrival time and priority. For example, the orange-colored PD thread indicates that it has higher priority, while the blue color indicates lower priority. When the PD4 with the highest priority is woken up, it will be inserted into the head of the ready queue, but

the PD3 with the lowest priority will be placed at the tail. After the PD thread gets the CPU and completes its job, it will go to sleep and wait for the OVS to wake up again. The strategy in this example makes S2H flexible to implement a differentiated latency guarantee. A simple test result is shown in Fig. 5.9, the test VM is set with a fixed sleep time threshold and highest priority. When the background traffic gets larger, the test VM under S2H can maintain the original latency of 30us, while the latency under the original vHost-User is increased by 50% to 45us.

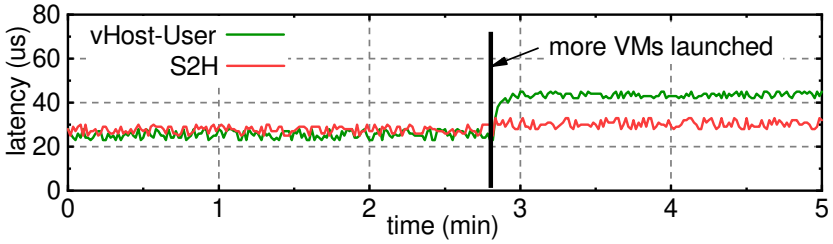


Figure 5.9 The differential latency under "batch-grained" scheduling

5.4.4.2 Overhead analysis

The main overhead of "batch-grained" scheduling comes from the context switching among PD threads. As the scheduling only changes the order of threads waiting on the PD cores, it does not affect the processing capacity of the PD cores. It is the context switching in thread scheduling that affects PD cores' processing capacity. We test the context switching times on a single PD core which serves different numbers of VMs with different sizes of packets to show how it affects the performance. The results of *perf* are shown in Table 5.3. The overhead of context switching is independent of the number of VMs. As the packet size increases, the effect of context switching becomes smaller. That is because each context switching only happens after each batch process. The time used for one context switching is fixed, while the time used for copying a batch of 1518-byte packets is far much more than copying a batch of 64-byte packets. So, when transferring 1518-byte packets, there will be much fewer times of context switching per second. The worst-case happens when copying 64-byte packets, and nearly 10% of the CPU is used for context switching, while the rate with 1518-byte packets drops to less than 1%. This overhead is acceptable for better isolation and flexibility.

In addition to these benefits, there is still a problem that may occur when we bind so many "SCHED_FIFO" scheduled PD threads to the same CPU core. The core cannot be scheduled to any other threads with the default scheduling policy, even if they belong to the kernel. For example, some system calls, like page fault interrupts, would run a small piece of codes on

Table 5.3 The context switching frequency under different configurations and workloads

Workload	2VMs	4VMs	8VMs
no traffic	1.512M/sec	1.524M/sec	1.454M/sec
64-byte packet	0.196M/sec	0.206M/sec	0.194M/sec
512-byte packet	0.045M/sec	0.043M/sec	0.043M/sec
1518-byte packet	0.017M/sec	0.017M/sec	0.017M/sec

each core of the physical server platform. This problem can be easily solved by isolating the PD cores in the *grub* file.

5.5 Evaluation

In Section 5.3, we have illustrated that S2H can provide two types of isolation — isolation among VMs and isolation between a VM and the host, which greatly reduces security risks. As the use of hypervisor and newly added shared-ring may introduce additional overheads, in this section, we will evaluate the performance of S2H and compare it with that of native vHost-User.

The performance evaluation and comparisons are from three perspectives. As our goal is to measure VNIO performance, we firstly evaluate and compare the VNIO data path performance of S2H with that of vHost-User. But considering the PD capacity in the VNIO data path is ultimately reflected in the application performance inside the VM, so we deploy various types of applications (e.g., IP lookup, TCP, Nginx and VM-to-VM communication) inside the VM to compare the application experience under the two architectures. Besides, to measure the scalability of a single VM and the entire physical server, we compare the performance in the multi-queue and multi-tenant scenarios under the two architectures.

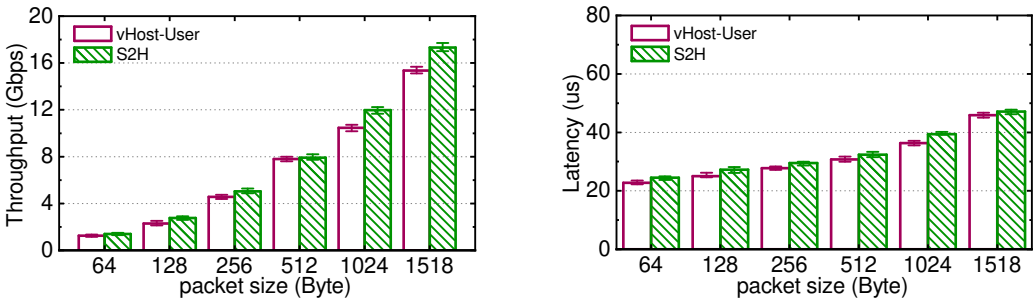
In the experimental setup, we use a server with two Xeon CPU E5-2640 v3 2.60GHz (2x8 cores and 2 logical cores in each physical core) for running the whole cloud platform for the two architectures. The other configurations of the server are as follows: 128GB DDR4 memory at 1866MHz, one Intel 82599ES 10-Gigabit Dual Port NICs, ubuntu 16.04.1 (kernel 4.8.0) as both host OS and guest OS, QEMU 2.10, DPDK 17.11.2 and OVS-2.9.2. Every VM is allocated with 2GB memory and one logical core for all tests.

We use TestCenter from Spirent[96] as traffic generator for the test of packet forwarding and use qperf on a directly connected server with the same configuration for the evaluation of the TCP performance. In all test scenarios, the VM configurations in the two architectures are

consistent, and we repeat 10 times of running experiments to eliminate the accidental errors. For S2H, the PD thread sleep time threshold is set little enough to achieve its peak performance.

5.5.1 Datapath performance

We first compare VNIO datapath performance by measuring the forwarding rate of VM on S2H with that of vHost-User. The DPDK driver is used inside a VM to ensure that the VM's internal network processing will not become a bottleneck. To evaluate the performance of the data path with PD running on a single core on vHost-User, OVS launches a PMD thread, which is bound to one logical core. For S2H, as the packet copying by the PD threads in QEMUs take most of the workload originally taken by the PMD thread on vHost-User, we also assign them with a logical core. However, although the OVS PMD thread in S2H only has very lighted look-up workload, it still needs to consume little CPU resources. So in our implementation, S2H requires a little more CPU resources than vHost-User in any case, because its architecture decouples flow table lookup and PD procedure. The experimental configuration follows RFC 2544[123] — throughput and latency are evaluated with zero packet loss. Different sizes of packets (64, 128, 256, 512, 1024, 1518 bytes) are generated by TestCenter and forwarded back via the VM internal DPDK l2fwd program.



(a) Throughput (b) Latency

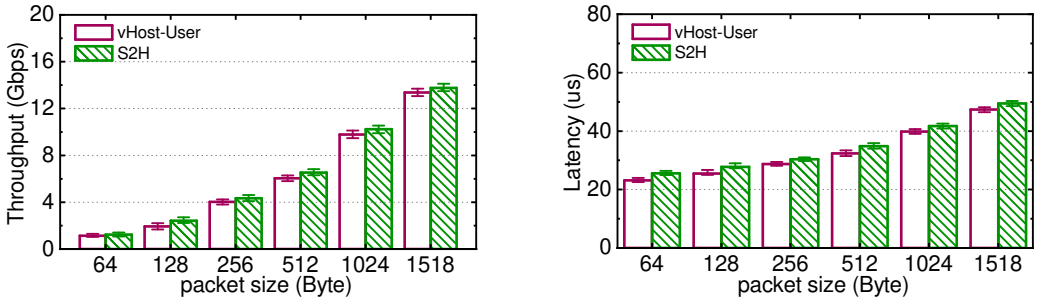
Figure 5.10 The performance comparison of datapath

Fig. 5.10(a) shows the datapath throughput of S2H and that of vHost-User. S2H achieves up to 14% throughput improvement with 1024-byte packet and 11% average throughput improvement compared to vHost-User. The throughput improvement of S2H is mainly due to two reasons, 1) a running PD thread cannot be preempted by any other threads thus avoiding the overhead due to unnecessary context switching, and 2) the little CPU resources used by the OVS PMD thread undertakes some flow table lookup workload. In Fig. 5.10(b), the latency of S2H increases by 2–9% under different packet sizes, up to 2μs compared to those of vHost-User. The increase

in latency is caused by the additional workload on the notification path. As mentioned in Section 5.4.2, we add one packet descriptor copying operation during each packet delivery. The increase in latency is constant and independent of the packet size.

5.5.2 Performance of Applications

For a more realistic study, we consider applications and the kernel protocol stack deployed in the VM of S2H and vHost-User. The performance of the applications including IP lookup, TCP, Nginx and VM-to-VM communication are evaluated.

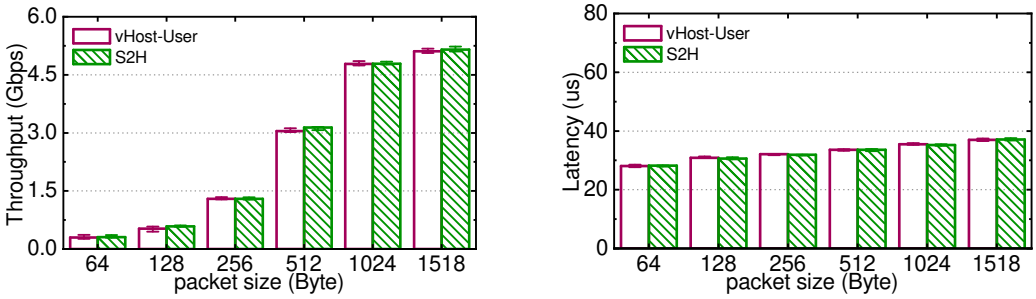


(a) Throughput (b) Latency

Figure 5.11 The performance comparison of IP lookup

IP lookup. Running Network NFV applications on the cloud platform is a common trend. To evaluate the performance of NFV in both architectures, we use VM with DPDK driver to enable high-performance IP lookup. Fig. 5.11(a) shows the throughput of IP lookup on S2H and vHost-User, respectively. S2H achieves up to 20% throughput improvement with 128-byte packet and 9% for average improvement compared to that using vHost-User. Fig. 5.11(b) shows the latency of IP lookup in S2H and vHost-User. The latency of S2H is 4–9% higher than that of vHost-User. The performance differences in the two implementations are caused, similarly, by CPU resources and the operations of shared-ring on S2H, as illustrated in VNIO datapath performance. There is a 10–15% decline in throughput compared to that on the datapath due to the workload of IP lookup in VMs.

TCP stream. TCP performance is an important indicator to measure the stability of VNIO. TCP bandwidth and latency are tested by qperf[77]. The qperf client runs in a VM and the qperf server runs in another direct-connected physical server. Fig. 5.12(a) shows the TCP bandwidth of S2H and vHost-User. S2H achieves up to 4% throughput improvement with 128-byte packet and 2% improvement on average compared to vHost-User. Fig. 5.12(b) shows the latency of TCP packet transmission by VM using S2H and vHost-User. The latency is relatively stable

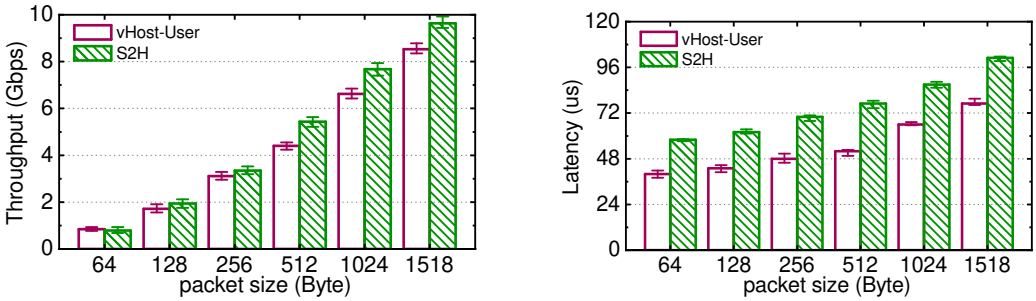


(a) TCP bandwidth (b) TCP latency

Figure 5.12 The comparison of TCP bandwidth and latency

and almost the same in the two implementations, as the performance in this case is mainly affected by packet processing in the VM kernel protocol stack.

Nginx. As a high-performance HTTP server and reverse proxy, Nginx [91] has been widely deployed on the cloud platform. We further evaluate the performance of Nginx on S2H and vHost-User. The Nginx 1.10.3 is deployed in one VM of S2H and vHost-User respectively. We use the Apache Bench running on another directly connected server to test the throughput and response time of Nginx on VM for HTTP requests. The throughput is 9733 responses per second for S2H and 9431 for vHost-User. The average latency per response is 208ms in S2H and 205ms in vHost-User. The performance of Nginx application on the two architectures is very close. The main reason is that the bottleneck appears at the VM kernel protocol stack and the Nginx application, which are configured the same for both architectures.



(a) VM-to-VM throughput (b) VM-to-VM latency

Figure 5.13 The performance comparison under VM-to-VM scenarios

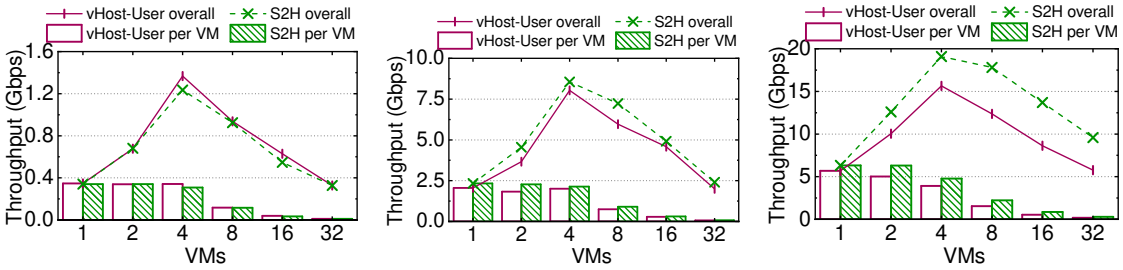
VM-to-VM. VM-to-VM communication is not very common on cloud tenant platforms. But in some other scenarios like NFV and distributed computing, VM-to-VM performance significantly affects the performance of user's applications. We test the VM-to-VM performance by using two VMs to form a service chain. The traffic generated by TestCenter is sent to one VM (say VM1) through NIC, and then forwarded to another VM (say VM2) through DPDK

l2fwd program. Finally, VM2 uses l2fwd program to forward it back to the TestCenter. As shown in Fig. 5.13(a), S2H achieves up to 10% throughput improvement with 128-byte packets and 7% for the average improvement compared to vHost-User. Due to the overhead of context switching, the S2H throughput in 64-byte packets is slightly worse than vHost-User. In Fig. 5.13(b), the latency of S2H increases by 22–30% under different packet sizes. The high latency in S2H VM-to-VM test is caused by the multiple times of packet descriptor copying during each packet's PD procedure and the synchronization overhead among shared *mbuf pool*. To alleviate the heavy overhead of S2H during the VM-to-VM communication, we can adopt a VM-to-VM fast-path[124, 125] to bypass the vSwitch and reduce times of memory copying.

5.5.3 Scalability

Multi-tenant and multi-queue scenarios of S2H are evaluated in this section.

Multi-tenants scalability. Scalability in the multi-tenant scenario is evaluated by the network I/O performance with the growing number of VMs on a physical server. To simulate a real cloud platform environment, we run up to 32 VMs on a server. These VMs are all configured with kernel drivers and set the forwarding rules using *iptables*[126], which can forward traffic back to the TestCenter. Two logical cores are assigned to PD procedures in both architectures to undertake the heavy packet copying workload.



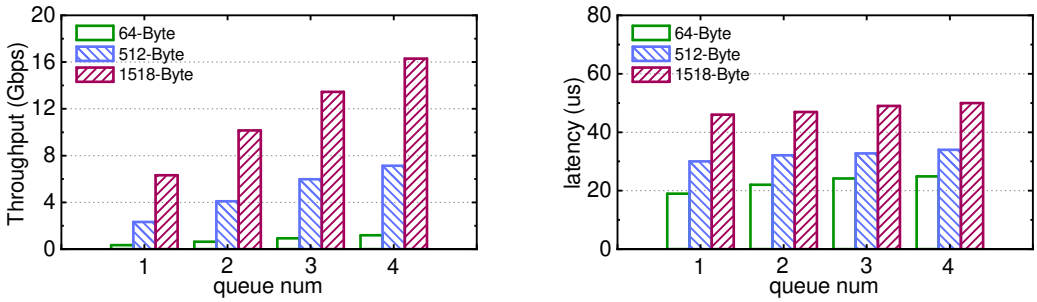
(a) 64-byte packet (b) 512-byte packet (c) 1518-byte packet

Figure 5.14 The scalability comparison in multi-tenant scenarios

The results are shown in Fig. 5.14. S2H achieves good scalability, especially in the case of large packets. Both S2H and vHost-User achieve the maximum overall throughput when running 4 VMs, where the throughput of S2H is 25% higher than that of vHost-User. As shown in the figure, the throughput per VM and the total throughput decrease with the number of running VMs being increased to 8 or more due to the competition of computing, memory and virtual network I/O resources. This is an inherent issue in OVS and it has nothing to do with this work.

Comparing these figures, it can be seen that the throughput of two architectures have different sensitivity with different packet sizes. The throughput of S2H is slightly worse than that of vHost-User when delivering 64-byte packets. But as packet size increases, S2H has an advantage over vHost-User. When running 8 VMs, S2H's total throughput is about 40% higher than that of vHost-User with 1518-byte packets and 20% higher with 512-byte packets. This is caused by the context switching of PD threads as illustrated in Section 5.4.2.

Multi-queue scalability. Multi-queue is a crucial feature to improve the packets processing ability inside a VM. We evaluate the scalability of multi-queue in the S2H architecture by continuously adding the number of queues in a single VM. The PD procedure is bound to one logical core, and the VM running kernel iptables NAT forwarding is given enough resources (one core for each queue).



(a) Throughput (b) Latency

Figure 5.15 The multi-queue performance of S2H

Fig. 5.15(a) shows the throughput of a single VM with the number of queues varying from 1 to 4. Each time one more queue is added, the VM throughput increases by an average of 0.76 times the single queue performance. The throughput increment decreases as the packet size goes down, for about 0.8 times of the single queue performance with 64-byte packets and only 0.67 times with 1518-byte packets. This is because the PD thread has more packets to deliver in one batch as the number of queues increases, and that greatly reduces the impact of context switching overhead, especially when the packet size is small. The latency of the architecture with different numbers of queues is shown in Fig. 5.15(b). Since the multi-queue scheme increases the processing capability of VM without introducing additional overheads on both data path and notification path, the latency does not change as the number of queues increases.

5.5.4 Summary of performance evaluation

Compared with the native vHost-User, the S2H prototype system achieves up to 11% improvement on data path throughput, but suffers from 2-9% latency increase. These performance fluctuations are negligible, and have no effect on the network applications where the processing bottleneck is inside the VM kernel protocol stack rather than VNIO. S2H also shows good scalability in multi-tenant scenarios. But as limited by the context switching overhead, S2H is at a little performance disadvantage compared to vHost-User in the case of delivering small packets. So in a word, the S2H prototype system is performance-friendly, which can maintain the high efficiency as the native vHost-User while guaranteeing memory isolation.

5.6 Conclusion

This chapter focuses on the secure isolation issues in VNIO mechanisms implemented by vSwitch. VNIO is an enabling technology in the context of cloud computing. Existing para-virtualized VNIO solutions often pursue the performance at the expensive cost of VM isolation. In this work, we classified existing para-virtualized solutions into S2V and V2S memory-sharing models and then analyzed their isolation issues. To solve this problem, we proposed a new S2H model, which shares the host-side I/O memory to the hypervisor. In order to adopt the S2H model in the VNIO design, we implemented an efficient shared memory access method which exploits the DPDK memory management to address the MPMC issue. In addition, we proposed a "batch-grained" scheduling strategy for PD threads to ensure network performance in multi-tenant scenarios. We integrated the *de-facto* software-based VNIO standard, vHost-User architecture into our prototype S2H system and evaluated its performance. The prototype exhibited a good trade-off between the isolation and the performance. It can achieve the VM isolation with the comparable throughput and less than 9% latency increase compared to the techniques based on the native vHost-User. The results also demonstrated the effectiveness of the proposed concurrent shared memory access and scheduling strategy in ensuring scalability.

Chapter 6

Conclusion

As cloud computing has become the trend and paradigm of service deployment, enterprises and individuals tend to purchase VMs on cloud platforms to deploy their own services. The deployment density of VMs on today's cloud platform servers is getting higher, and tenants' requirements for isolation are also increasing. However, the vSwitch, which provides network forwarding and connection for VMs, has become a high-privileged user-space component with resource sharing and intensive I/O operations in the continuous pursuit of higher performance. While these evolutions provide tenants with efficient virtual networks, they also bring potential isolation issues. Specifically, due to the sharing of physical resources, the competition among tenants makes the traditional QoS methods can no longer guarantee tenant's network performance isolation; due to the sharing of data structures, malicious tenants can create common failure points and form DoS attacks to interfere other tenants; finally, in terms of I/O operation and memory security, the existing VNIO using shared memory for acceleration brings the secure isolation risk of illegal memory access. Therefore, how to design effective mechanisms in the vSwitch to ensure VM isolation in terms of network performance, network failure and security is critical to the cloud platforms.

In order to realize the network performance isolation for tenants, this paper proposes a CPU-cycle isolation based network QoS guarantee method (C2QoS). Aiming at the volatile processing capability of CPU cores in vSwitch, C2QoS establishes a model between network performance and CPU resource consumption through the measurement-driven method, and then allocates I/O-dedicated CPU resources to VMs according to the model. To ensure the isolation on CPU resources, C2QoS implements a C2TB mechanism, which performs packet forwarding tasks for each VM according to the available CPU resources, so as to ensure VM network bandwidth isolation. To solve the competition of CPU resources in timing, C2QoS im-

plements an HBS scheduling mechanism to realize differentiated latency through priority-based forwarding task scheduling. The experimental results on the OVS-DPDK platform prove that, compared with the existing packet/flow-based QoS guarantee methods, the C2QoS method proposed in this paper can achieve strict tenant network bandwidth isolation with 2% CPU overhead. At the same time, the additional latency caused by competition is reduced by 80%.

In order to realize the network fault isolation for tenants, this paper proposes a flow table isolation based data plane DoS attack defense mechanism (D-TSE). For the failure caused by the malicious tenant on the classification of the shared flow table in vSwitch, D-TSE attributes the attack to the sharing of data structure and processing procedure through experiments. To solve the isolation issue caused by the sharing of flow table structure, D-TSE implements a separated three-level flow table architecture to isolate possible common failure points. In order to break the paradox of needing to predict the belongings of packets before classification, D-TSE designed a lightweight PRECLS module before flow tables to realize fast packet redirection. For efficiency, D-TSE designs a batch re-aggregation mechanism to ensure the performance of packet forwarding after PRECLS redirection. The experimental results on the OVS-DPDK platform show that, compared with the native non-isolation system, the D-TSE mechanism proposed in this paper achieves the tenant network failure isolation at the cost of no more than a 5% performance drop.

In order to realize the security isolation for tenants, this paper proposes a memory access isolation based VNIO mechanism (S2H). Aiming at the security isolation introduced by the shared memory between VM and vSwitch in the existing VNIO solutions, S2H proposes a secure memory sharing model by introducing hypervisor into the data path. According to the model, a secure para-virtualized network I/O mechanism – S2H, is proposed to be compatible with the existing cloud platform infrastructure. For efficiency, this paper uses an efficient concurrent shared memory framework to solve the MPMC problem, and proposes a "batch-grained" thread scheduling method to save CPU resources. By implementing the prototype system of S2H mechanism on the QEMU/KVM and OVS-DPDK platforms, this paper fully verified its effectiveness. Compared with the widely used vHost-User mechanism, the S2H mechanism guarantees the secure isolation of VM memory at the cost of only a 2–9% increase in latency, while keeping the same performance and scalability.

On the basis of the above research content, we will continue to carry out research in the following two aspects in the future:

1). Realizing the isolation among tenants in the vSwitch accelerated by the smart NICs. When software-based vSwitch reaches the performance ceiling, smart NIC acceleration has become

an inevitable trend in the development of cloud networks. Under the co-design of software and hardware, the packet processing procedure becomes more complex and unpredictable. In the next step, we plan to use the methodology in existing research to achieve tenant isolation in cloud network for this complex scenario.

2). Congestion control and back-pressure mechanism under cloud-scale overlay network. Although the works in this paper have achieved network performance guarantee for tenants through isolation mechanisms, these methods can only solve the contention and congestion of packets in the vSwitch processing stage. However, some of the congestion situations in the cloud network occur in the NIC receiving packets, which leads to the problem of unfair packet loss in the NIC queue. To make matters worse, the network configuration and routing in the overlay network are more complex, and the existing mechanisms such as DCQCN and ECN cannot be directly applied to cloud-scale virtual networks. So our next plan is to design a set of congestion control and backpressure mechanisms for the congestion that occurs in virtual networks to achieve self-healing for congestion.

Bibliography

[1] Amazon, "Aws." https://aws.amazon.com, 2021.

[2] Microsoft, "Azure." https://azure.microsoft.com, 2021.

[3] Alibaba Group, "Alibabacloud." https://www.alibabacloud.com, 2021.

[4] Google, "Googlecloud." https://cloud.google.com, 2021.

[5] F. Bellard, "QEMU: Open source processor emulator." https://www.qemu.org/, 2021.

[6] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, pp. 225–230, Dttawa, Dntorio, Canada, 2007.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 164–177, ACM, 2003.

[8] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Computer Networks*, vol. 54, no. 5, pp. 862–876, 2010.

[9] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, and et al, "Network virtualization in multi-tenant datacenters," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 203–216, USENIX Association, 2014.

[10] M. Kerrisk, "Bridge–linux manual page." https://man7.org/linux/man-pages/man8/bridge.8.html, 2021.

[11] Linux Foundation, "Data Plane Development Kit (DPDK)." https://www.dpdk.org, 2021.

[12] P. P. Tang and T. . C. Tai, "Network traffic characterization using token bucket model," in *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*, pp. 51–62, IEEE Computer Society, 1999.

[13] M. Kerrisk, "Hierarchy token bucket in linux kernel." https://www.man7.org/linux/man-pages/man8/tc-htb.8.html, 2021.

[14] M. Devera, "Htb." http://luxik.cdi.cz/~devik/qos/htb/, 2021.

[15] K. To, D. Firestone, G. Varghese, and J. Padhye, "Measurement based fair queuing for allocating bandwidth to virtual machines," in *Proceedings of the ACM SIGCOMM Workshop on Hot topics in Middleboxes and Network Function Virtualization (HotMiddlebox@SIGCOMM)*, pp. 14–19, ACM, 2016.

[16] J. R. Davin and A. T. Heybey, "A simulation study of fair queueing and policy enforcement," *ACM SIGCOMM Computer Communication Review*, vol. 20, no. 5, pp. 23–29, 1990.

[17] M. Shreedhar and G. Varghese, "Efficient fair queueing using deficit round robin," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pp. 231–242, ACM, 1995.

[18] J. C. R. Bennett and H. Zhang, "WF^2Q: Worst-case fair weighted fair queueing," in *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*, pp. 120–128, IEEE Computer Society, 1996.

[19] J. Mace, P. Bodik, M. Musuvathi, *et al.*, "2DFQ: Two-dimensional fair queuing for multi-tenant cloud services," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pp. 144–159, ACM, 2016.

[20] J. Corbet, "qdisc." https://lwn.net/Articles/564978/, 2021.

[21] V. Addanki, L. Linguaglossa, J. Roberts, and D. Rossi, "Controlling software router resource sharing by fair packet dropping," in *Proceedings of the 17th International IFIP Networking Conference (Networking)*, pp. 388–396, IFIP, 2018.

[22] P. Kumar, N. Dukkipati, N. Lewis, *et al.*, "Picnic: predictable virtualized NIC," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pp. 351–366, ACM, 2019.

[23] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, p. 65–77, ACM, 2021.

[24] D. Kreutz, F. M. V. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pp. 55–60, ACM, 2013.

[25] R. Kandoi and M. Antikainen, "Denial-of-service attacks in openflow SDN networks," in *Proceedings of the 14th IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pp. 1322–1326, IEEE, 2015.

[26] Y. Qian, W. You, and K. Qian, "Openflow flow table overflow attacks and countermeasures," in *Proceedings of European Conference on Networks and Communications (EuCNC)*, pp. 205–209, IEEE, 2016.

[27] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: detecting security attacks in software-defined networks," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*, The Internet Society, 2015.

[28] L. Csikor, D. M. Divakaran, M. S. Kang, A. Korösi, B. Sonkoly, D. Haja, D. P. Pezaros, S. Schmid, and G. Rétvári, "Tuple space explosion: a denial-of-service attack against a software packet classifier," in *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pp. 292–304, ACM, 2019.

[29] L. Csikor, V. Ujawane, and D. M. Divakaran, "On the feasibility and enhancement of the tuple space explosion attack against open vswitch," *arXiv*, vol. 2011.09107, 2020.

[30] L. Csikor, M. Szalay, G. Rétvári, G. Pongrácz, D. P. Pezaros, and L. Toka, "Transition to SDN is HARMLESS: hybrid architecture for migrating legacy ethernet switches to SDN," *IEEE/ACM Transactions on Networking*, vol. 28, no. 1, pp. 275–288, 2020.

[31] Linux Foundation, "DPDK vHost User Ports." https://docs.openvswitch.org/en/latest/topics/dpdk/vhost-user/, 2021.

[32] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky, "Efficient and scalable paravirtual i/o system," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 231–242, USENIX Association, 2013.

[33] A. Menon, A. L. Cox, and W. Zwaenepoel, "Optimizing network virtualization in xen," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 15–28, USENIX Association, 2006.

[34] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: high performance and flexible networking using virtualization on commodity platforms," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 445–458, USENIX Association, 2014.

[35] A. C. Macdonell *et al.*, *Shared-memory optimizations for virtual machines*. University of Alberta Edmonton, Canada, 2011.

[36] C. MacDonell, "Nahanni, a shared memory interface for kvm." http://www.linux-kvm.org/images/e/e8/0.11.Nahanni-CamMacdonell.pdf, 2021.

[37] J. R. Santos, Y. Turner, G. J. Janakiraman, and I. Pratt, "Bridging the gap between software and hardware techniques for i/o virtualization," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 29–42, USENIX Association, 2008.

[38] K. K. Ram, J. R. Santos, and Y. Turner, "Redesigning xen's memory sharing mechanism for safe and efficient I/O virtualization," in *Proceedings of the 2nd Conference on I/O Virtualization*, pp. 1–1, USENIX Association, 2010.

[39] MITRE Corporation, "CVE-2018-1059." https://www.cvedetails.com/cve/CVE-2018-1059/, 2018.

[40] P. Stewin and I. Bystrov, "Understanding DMA malware," in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 21–41, Springer, 2012.

[41] N. Amit, M. Ben-Yehuda, D. Tsafrir, and A. Schuster, "vIOMMU: efficient iommu emulation," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 73–86, USENIX Association, 2011.

[42] J. Wang and P. Xu, "Vhost and vIOMMU." http://www.linux-kvm.org/images/c/c5/03x07B-Peter_Xu_and_Wei_Xu-Vhost_with_Guest_vIOMMU.pdf, 2016.

[43] E. Auger, "vIOMMU/ARM: full emulation and virtio-iommu approaches." https://www.linux-kvm.org/images/8/8e/Viommu_arm.pdf, 2017.

[44] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, vol. 2016, no. 86, pp. 1–118, 2016.

[45] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska, "S-nfv: Securing nfv states by using sgx," in *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pp. 45–48, ACM, 2016.

[46] O. Oleksenko, B. Trach, R. Krahn, A. Martin, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX enclaves from practical side-channel attacks," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 227–240, USENIX Association, 2018.

[47] Y. Yang, H. Jiang, Y. Wu, Y. Lv, X. Li, and G. Xie, "C2QoS: CPU-Cycle based network QoS strategy in vswitch of public cloud," in *Proceedings of the 17th IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pp. 438–444, IFIP/IEEE, 2021.

[48] Y. Yang, H. Jiang, Y. Wu, C. Han, Y. Lv, X. Li, B. Yang, S. Fdida, and G. Xie, "C2QoS: Network QoS guarantee in vswitch through CPU-cycle management," *Journal of Systems Architecture*, vol. 116, p. 102148, 2021.

[49] Y. Yang, H. Jiang, Y. Liang, Y. Wu, Y. Lv, X. Li, and G. Xie, "Isolation guarantee for efficient virtualized network i/o on cloud platform," in *Proceedings of the 22nd IEEE International Conference on High Performance Computing and Communications (HPCC)*, pp. 344–351, IEEE, 2020.

[50] Y. Yang, H. Jiang, G. Zhang, X. Wang, Y. Lv, X. Li, S. Fdida, and G. Xie, "S2H: Hypervisor as a setter within Virtualized Network I/O for VM isolation on cloud platform," *Computer Networks*, vol. 201, p. 108577, 2021.

[51] Microsoft, "Azure for operators." https://azure.microsoft.com/en-us/industries/telecommunications/#overview, 2021.

[52] Amazon, "AWS abandoned Xen and switched to KVM." https://aws.amazon.com/cn/ec2/faqs/#compute-optimized, 2021.

[53] M. Krasnyansky, "Universal TUN/TAP device driver." https://www.kernel.org/doc/html/latest/networking/tuntap.html, 2021.

[54] N. McKeown, T. E. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. S. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[55] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu, "Understanding performance interference of I/O workload in virtualized cloud environments," in *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*, pp. 51–58, IEEE Computer Society, 2010.

[56] L. Cherkasova and R. Gardner, "Measuring CPU overhead for I/O processing in the xen virtual machine monitor," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 387–390, USENIX Association, 2005.

[57] L. Rizzo, "netmap: A novel framework for fast packet I/O," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 101–112, USENIX Association, 2012.

[58] Linux Foundation, "Vector Packet Processor (VPP)." https://fd.io/docs/vpp/master/, 2021.

[59] Linux Foundation, "Open vSwitch (OVS)." http://www.openvswitch.org/, 2021.

[60] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, *et al.*, "The design and implementation of open vswitch," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 117–130, USENIX Association, 2015.

[61] L. Rizzo and G. Lettieri, "Vale, a switched ethernet for virtual machines," in *Proceedings of the 8th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, pp. 61–72, ACM, 2012.

[62] S. Han, "Berkeley Extensible Software Switch (BESS)." https://github.com/NetSys/bess/wiki/BESS-Overview, 2021.

[63] M. Paolino, N. Nikolaev, J. Fanguede, and D. Raho, "Snabbswitch user space virtual switch benchmark and performance optimization for NFV," in *Proceedings of the IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pp. 86–92, IEEE, 2015.

[64] D. Firestone, "VFP: A virtual switch platform for host SDN in the public cloud," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 315–328, USENIX Association, 2017.

[65] J. Heinanen and R. Guerin, "RFC2697(srTCM)." https://www.rfc-editor.org/rfc/rfc2697.html, 2021.

[66] J. Heinanen and R. Guerin, "RFC2698(trTCM)." https://www.rfc-editor.org/rfc/rfc2698.html, 2021.

[67] Linux Foundation, "DPDK traffic metering." http://doc.dpdk.org/guides/prog_guide/traffic_metering_and_policing.html, 2021.

[68] A. Saeed, N. Dukkipati, V. Valancius, *et al.*, "Carousel: Scalable traffic shaping at end hosts," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pp. 404–417, ACM, 2017.

[69] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, *et al.*, "Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, USENIX Association, 2018.

[70] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pp. 135–146, ACM, 1999.

[71] R. Russell, "virtio: towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[72] KVM, "Usingvhost." https://www.linux-kvm.org/index.php?title=UsingVhost&oldid=3513, 2021.

[73] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the art of network function virtualization," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 459–473, USENIX Association, 2014.

[74] F. Checconi, L. Rizzo, and P. Valente, "QFQ: Efficient packet scheduling with tight guarantees," *IEEE/ACM Transactions on Networking*, vol. 21, no. 3, pp. 802–816, 2013.

[75] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, *et al.*, "Programmable packet scheduling at line rate," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pp. 44–57, ACM, 2016.

[76] A. Saeed, Y. Zhao, N. Dukkipati, E. Zegura, *et al.*, "Eiffel: efficient and flexible software packet scheduling," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 17–32, USENIX Association, 2019.

[77] J. George, "qperf." https://linux.die.net/man/1/qperf, 2021.

[78] The Kernel Development Community, "What is numa." https://www.kernel.org/doc/html/latest/vm/numa.html, 2021.

[79] The Kernel Development Community, "Numa locality." https://www.kernel.org/doc/html/latest/admin-guide/mm/numaperf.html, 2021.

[80] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference (IMC)*, pp. 267–280, ACM, 2010.

[81] A. Morton, "Imix genome: Specification of variable packet sizes for additional testing." https://tools.ietf.org/html/rfc6985, 2021.

[82] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive, "A flexible model for resource management in virtual private networks," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pp. 95–108, ACM, 1999.

[83] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, "Flurries: Countless fine-grained NFs for flexible per-flow customization," in *Proceedings of the 12th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pp. 3–17, ACM, 2016.

[84] I. Stoica, H. Zhang, and T. S. Eugene Ng, "A hierarchical fair service curve algorithm for link-sharing, real-time and priority services," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pp. 249–262, ACM, 1997.

[85] T. K. Wignall, "Priority queuing systems with and without feedback," *Operations Research*, vol. 21, no. 3, pp. 764–776, 1973.

[86] A. Derbala, "Priority queuing in an operating system," *Computers & Operations Research*, vol. 32, pp. 229–238, 2005.

[87] P. Chuprikov, S. Nikolenko, and K. Kogan, "Priority queueing with multiple packet characteristics," in *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*, pp. 1418–1426, IEEE, 2015.

[88] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 455–468, USENIX Association, 2015.

[89] Intel Corporation, "Intel 64 and ia-32 architectures software developer's manual," *Volume 3A: System Programming Guide, Part*, vol. 1, no. 64, p. 64, 64.

[90] C. Evans, "Vsftpd: Probably the most secure and fastest ftp server for unix-like systems." https://security.appspot.com/vsftpd.html, 2021.

[91] I. Sysoev, "Nginx." http://nginx.org/, 2021.

[92] J. Dugan, S. Elliott, B. A. Mah, and et al, "iperf." https://iperf.fr/, 2021.

[93] W. Glozer, "wrk." https://github.com/wg/wrk, 2021.

[94] J. Cao, M. Xu, Q. Li, K. Sun, Y. Yang, and J. Zheng, "Disrupting SDN via the data plane: A low-rate flow table overflow attack," in *Proceedings of the 13th International Conference on Security and Privacy in Communication Systems (SecureComm)*, pp. 356–376, Springer, 2017.

[95] L. Csikor, C. Rothenberg, D. P. Pezaros, S. Schmid, L. Toka, and G. Rétvári, "Policy injection: A cloud dataplane dos attack," in *Proceedings of the ACM SIGCOMM Conference on Posters and Demos*, pp. 147–149, ACM, 2018.

[96] Spirent Communications, "Testcenter—verifying network and cloud evolution." https://www.spirent.com/products/testcenter-ethernet-ip-cloud-test, 2021.

[97] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (N. Feamster and J. C. Mogul, eds.), pp. 371–384, USENIX Association, 2013.

[98] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," in *Proceedings of the 16th International Conference on High-performance Computer Architecture (HPCA)*, pp. 1–10, IEEE Computer Society, 2010.

[99] R. L. Solomon and T. E. Hoglund, "Paravirtualization acceleration through single root i/o virtualization," 2012. US Patent 8,332,849.

[100] A. Leivadeas, M. Falkner, and N. Pitaev, "Analyzing service chaining of virtualized network functions with SR-IOV," in *Proceedings of the 21st IEEE International Conference on High Performance Switching and Routing (HPSR)*, pp. 1–6, IEEE, 2020.

[101] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman, "Direct device assignment for untrusted fully-virtualized virtual machines," *Technical Report H-0263*, 2008.

[102] E. Zhai, G. D. Cummings, and Y. Dong, "Live migration with pass-through device for linux VM," in *Proceedings of the Ottawa Linux Symposium (OLS)*, pp. 261–268, 2008.

[103] B. Asvija, R. Eswari, and M. B. Bijoy, "Security in hardware assisted virtualization for cloud computing - state of the art issues and challenges," *Computer Networks*, vol. 151, pp. 68–92, 2019.

[104] Y. Kuperman, E. Moscovici, J. Nider, R. Ladelsky, A. Gordon, and D. Tsafrir, "Paravirtual remote I/O," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 49–65, ACM, 2016.

[105] X. Zhang, X. Zheng, Z. Wang, H. Yang, Y. Shen, and X. Long, "High-density multitenant bare-metal cloud," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, p. 483–495, ACM, 2020.

[106] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, L. Iannone, and J. Roberts, "Comparing the performance of state-of-the-art software switches for NFV," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT)*, pp. 68–81, ACM, 2019.

[107] M. Bourguiba, K. Haddadou, and G. Pujolle, "Packet aggregation based network I/O virtualization for cloud computing," *Computer Communications*, vol. 35, no. 3, pp. 309–319, 2012.

[108] L. Rizzo, G. Lettieri, and V. Maffione, "Speeding up packet I/O in virtual machines," in *Proceedings of the Symposium on Architecture for Networking and Communications Systems (ANCS)*, pp. 47–58, IEEE Computer Society, 2013.

[109] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, pp. 124–131, IEEE, 2009.

[110] MITRE Corporation, "Openvswitch CVE vulnerability statistics." https://www.cvedetails.com/vendor/12098/Openvswitch.html, 2017.

[111] MITRE Corporation, "CVE-2016-2074." https://www.cvedetails.com/cve/CVE-2016-2074/, 2016.

[112] S. Sreenivasamurthy and E. L. Miller, "SIVSHM: Secure inter-vm shared memory," Tech. Rep. UCSC-SSRC-16-01, University of California, Santa Cruz, May 2016.

[113] D. Wang, B. Hua, L. Lu, H. Zhu, and C. Liang, "Zcopy-vhost: Eliminating packet copying in virtual network I/O," in *Proceedings of the 42nd IEEE Conference on Local Computer Networks (LCN)*, pp. 632–639, IEEE Computer Society, 2017.

[114] K. K. Ram, A. L. Cox, M. Chadha, and S. Rixner, "Hyper-switch: A scalable software virtual switching architecture," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 13–24, USENIX Association, 2013.

[115] Intel Corporation, "EPT and VT-d." https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html?wapkw=vt, 2021.

[116] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress data path: fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pp. 54–66, ACM, 2018.

[117] M. Fleming, "A thorough introduction to eBPF." https://lwn.net/Articles/740157/, 2017.

[118] W. Tu, Y. Wei, G. Antichi, and B. Pfaff, "revisiting the open vswitch dataplane ten years later," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pp. 245–257, ACM, 2021.

[119] ARM, "TrustZone." https://developer.arm.com/technologies/trustzone, 2021.

[120] Y. Cheng, X. Ding, and R. H. Deng, "DriverGuard: Virtualization-based fine-grained protection on I/O flows," *ACM Transactions on Information and System Security (TISSEC)*, vol. 16, no. 2, pp. 1–30, 2013.

[121] MITRE Corporation, "CVE-2019-14835." https://www.cvedetails.com/cve/CVE-2019-14835/, 2019.

[122] R. Yang, X. Chang, J. V. Misic, and V. B. Misic, "Performance modeling of linux network system with open vswitch," *Peer Peer Netw. Appl.*, vol. 13, no. 1, pp. 151–162, 2020.

[123] S. Bradner, "Benchmarking methodology for network interconnect devices," *RFC2544*, 1999.

[124] W. Wang, "Design of vhost-pci." http://www.linux-kvm.org/images/5/55/02x07A-Wei_Wang-Design_of-Vhost-pci.pdf, 2016.

[125] W. Wang, "Github link of vhost-pci." https://github.com/wei-w-wang/vhost-pci, 2017.

[126] R. Russell, "iptables." https://linux.die.net/man/8/iptables, 2021.