Hindawi
Wireless Communications and Mobile Computing
Volume 2018, Article ID 6263153, 14 pages
https://doi.org/10.1155/2018/6263153

Research Article

Empirical Experience and Experimental Evaluation of Open5GCore over Hypervisor and Container

Hung-Cheng Chang, Bo-Jun Qiu, Jyh-Cheng Chen, Tze-Jie Tan, Ping-Fan Ho, Chen-Hao Chiu, and Bao-Shuh Paul Lin

Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan

Correspondence should be addressed to Jyh-Cheng Chen; jcchen@ieee.org

Received 15 July 2017; Revised 8 November 2017; Accepted 19 December 2017; Published 21 January 2018

Academic Editor: Christos Verikoukis

One of the most important technologies for future 5G networks is to utilize Network Function Virtualization (NFV) to virtualize the network components. NFV provides flexibility, short time to market, and low cost solution to build network services, which are important features of 5G networks. Although the idea of virtualization is just being applied to cellular networks, it has been used in the community of cloud computing. There are two main virtualization techniques, hypervisor and container. In this paper, we present our practical experience of virtualizing Open5GCore, a commercial product of SDN-enabled Evolved Packet Core (EPC), over hypervisor and container. In addition to describing how to virtualize Open5GCore, we also present the experimental performance evaluation of the systems. Finally, some important lessons learned are provided.

1. Introduction

Cellular networks have been evolved to 4th generation (4G). One of the most important technologies for future 5G [1] networks is to utilize Network Function Virtualization (NFV) [2] to virtualize the network components in the 4G core network which is called Evolved Packet Core (EPC) [3]. NFV enables operators to virtualize hardware resources. It also makes special-purpose network equipment toward software solutions, that is, Virtualized Network Function (VNF) instances. A VNF instance can run on several virtual machines (VMs) which can be scaled out/in to adjust the VNF's computing and networking capabilities in real time. NFV provides flexibility, short time to market, and low cost solution to build the network services [2, 4–6], which are important features of 5G networks.

Although the idea of virtualization is just being applied to cellular networks, it has been used in the community of cloud computing. There are two main virtualization techniques, hypervisor [7] and container [8]. The hypervisor virtualizes the hardware and creates VMs to let multiple Operating Systems (OSs) run on the same hardware at the same time. With VMs, OSs can be cloned, deleted, or moved to another hardware. The container further virtualizes OS so there can be multiple isolated user-space instances of the host OS on the same hardware. Such container-based virtualization technology eliminates the overhead of running multiple OSs on a single machine. Therefore, container provides a lighter way for virtualization than hypervisor does.

There are some studies that evaluate the NFV performance over hypervisor and container such as the virtualization of HTTP proxy [9] and WebRTC [10]. However, there are very few articles discussing the virtualization of EPC using hypervisor and container. In this paper, we present our practical experience of virtualizing Open5GCore [11], a commercial product of SDN-enabled EPC, over hypervisor and container. In addition to describing how to virtualize Open5GCore, we also present the experimental performance evaluation of the systems. Finally, some important lessons learned are provided. This paper could be a good reference for anyone interested in virtualization of future 5G networks. The contributions of this paper include the following:

(i) To the best of our knowledge, we are the first reporting comprehensive experimental results of hypervisor and container over a commercial carrier-grade EPC to the public domain.

(ii) We discuss the security issues of EPC virtualization. Specifically, we found that Docker must work in *privileged mode* to set the network interfaces. However, setting it to privileged mode causes security issues.

(iii) The installations of Open5GCore on physical machines, hypervisor, and container are tedious and sometimes troublesome if there is no script to speed up the setup process. The lessons we learned can save time for others.

The rest of this paper is organized as follows. In Section 2, we provide background of virtualization techniques and Open5GCore. In Section 3, we discuss related work. The details of virtualizing of Open5GCore are presented in Section 4, followed by experimental results illustrated in Section 5. Section 6 summarizes the paper and provides the lessons learned.

2. Background

2.1. Virtualization Techniques. Traditionally, a physical machine allows exactly one OS running on it. With nowadays powerful computing capability, it is often not efficient to let only one OS to utilize the hardware resource. With the virtualization techniques, many OSs can share the same hardware [12]. That is, multiple OSs can run on a single physical machine and each OS view itself owns the hardware. Virtualization techniques come with the following advantages [7, 8, 12–14]:

(i) Resource sharing and high utilization of the resources: with multiple OSs running on a single physical machine, each hardware component could be efficiently used. All of the hardware resources are shared

(ii) Cost reduction: the resource sharing means that the same physical machine can run different tasks at the same time. The same hardware resources are distributed to all tasks. Thus, expenses for the hardware can be reduced

(iii) Ease for maintenance: it is more convenient to manage several services/tasks on a single physical machine. Furthermore, the independence of each VM on the same hardware makes system cloning and migration easier. The clone allows the services to be recovered in a very short time. The migration can move the services to another machine

In the following, we first briefly present the overview of *hypervisor* and *container*.

2.1.1. Hypervisor. Hypervisor is one major type of virtualization techniques. There are two major types of hypervisor, *native* hypervisor and *hosted* hypervisor. As shown in Figure 1, the native hypervisor lies directly over the hardware to control the hardware, so its performance is better. However, hardware support is needed. The hosted hypervisor runs on an OS called host OS. The performance is not as good as that of native hypervisor. However, it is more flexible because it

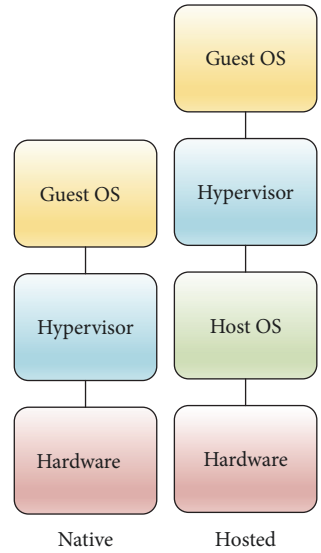


Figure 1: Two types of hypervisor.

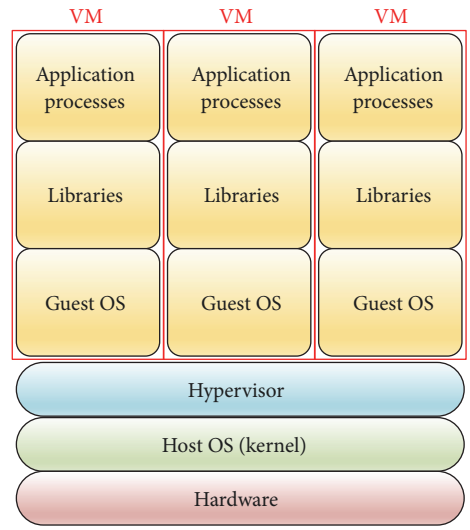


Figure 2: The virtualization structure of KVM.

can run on a conventional OS and hardware. In Figure 1, there can be multiple guest OSs. The native hypervisor includes Xen [15], VMWare [16] version 5.5 and its later versions, Virtual PC 2005 [17], VirtualBox [18], and Kernel-based Virtual Machine (KVM) [19]. The hosted hypervisor includes Virtual PC 2004 [17], VMWare [16] before version 5.5, and KVM [19]. Please note that KVM is a kernel module that makes the Linux kernel be a hypervisor and it uses QEMU [20] to create virtual machines. KVM lies directly over the hardware and meanwhile Linux is an OS itself. Therefore, KVM holds the characteristics of both native and hosted hypervisors.

2.1.2. Container. Container is another kind of popular virtualization techniques. Figures 2 and 3 show the difference between hosted hypervisor and LXD [21], one type of container. LXD excludes the guest OS. That is, a host OS

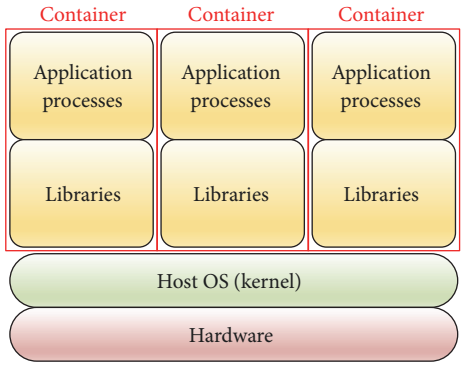


Figure 3: The virtualization structure of LXD.

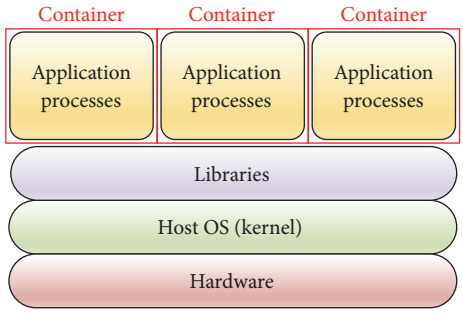


Figure 4: The virtualization structure of Docker.

allows the existence of multiple isolated user-space instances. Therefore, a container can run and execute faster than a VM. On the other hand, a container cannot perform the tasks which run in kernel space.

There are two main container-based virtualization techniques, LXD and Docker [22]. Figures 3 and 4 illustrate the virtualization structures of LXD and Docker, respectively. Compared with LXD, Docker further reduces to only the application level of a system, so the containers of Docker can run with very high performance and very little resource.

2.2. Virtual Network Interfaces. The Linux kernel provides many kinds virtualization techniques for network interfaces. With these techniques, one or more virtualized network interfaces, known as network interface cards (NIC), can be created for a physical machine, a virtual machine, or a container. Because either virtual machines or containers have their own physical interfaces, virtual interfaces are important for them to get link with other machines. Each node in Open5GCore needs two or three interfaces to communicate with other nodes. For virtualized Open5GCore, it is necessary to give virtual network interfaces to virtual machines or containers if Open5GCore is on the virtual machine or container. *veth* and the *macvlan/macvtap* are the commonly used interface virtualization techniques. We discuss them in this section.

2.2.1. The veth Interfaces. *veth* is the default virtual interface technique used in bridge mode of Docker. A veth device always appears in pair; that is, there are two interfaces existing separately in two sides of a veth device. Normally, one side of a veth pair is put in a container, and another side is put in the physical machine (host). Every time when data arrive at either side of a veth pair, the data are sent to another side directly. For a veth device, it is just like the loopback interface of Linux. Figure 6 illustrates an example of how veth works. Two pairs of veth devices are created in a host. One side of each veth (eth0) is connected to an individual container, and another side is connected to a bridge device in the host. Whenever communicating with other container or extranet via the physical interface (eth), packets need to go through the two sides of the veth device.

2.2.2. Interfaces of Macvlan/Macvtap. *macvlan* is a mechanism to create one or more virtual interfaces on a physical machine. Each virtual interface looks like an individual interface that connects to a switch with the physical one. Unlike veth devices that share the same MAC address, a *macvlan* device owns a unique MAC address. Therefore, *macvlan* can perform more operations on the MAC layer. *macvtap* is essentially based on the *macvlan* technique. It further provides the *device file* mechanism in Linux for each virtual interface. With the *device file*, the system can manage the virtual interfaces more easily.

There are two modes of the *macvlan/macvtap*: bridge and VEPA (Virtual Ethernet Port Aggregator). In the bridge mode shown in Figure 7, all of the virtual *macvlan/macvtap* interfaces are connected to a virtual switch. Communication between virtual interfaces is only inside the virtual switch. Only when the virtual interface communicates with the physical interface or extranet will the network flow reach the physical switch. Figure 8 illustrates the VEPA mode, in which communication between virtual interfaces is through the physical switch as if virtual interfaces are connected to the physical switch directly. In this case, communication between virtual interfaces can utilize the physical switch services that the virtual switch of *macvtap* cannot provide, such as network management, Spanning Tree Protocol (STP), and multicast.

Because *macvtap* performs better than veth [26], we use *macvtap* as the main interface virtualization technique in this paper.

2.3. Open5GCore. Open5GCore [11], developed by Fokus [27], is a commercial carrier-grade core network implementation. In this section, we first discuss why we choose Open5GCore.

Table 1 shows the testbed comparison. For core network, both nwEPC [24] and openair-cn [25] are open-source software. However, the development of nwEPC is frozen now. The most widely used open-source platform nowadays is openair-cn. However, it is a simple implementation of 3GPP R9/R10. Because the main purpose of openair-cn is to work with the eNB implementation of OpenAirInterface (OAI) [28], the core network cannot work with the commercial eNBs we tested. The reason is that many functions in openair-cn are not implemented yet. In addition, it is not very flexible and cannot reduce disk and memory usages compared with commercial products such as OpenEPC [23] and Open5GCore. OpenEPC, a 4G implementation, is

Table 1: Testbed comparison.

Testbed	Open source	Completeness	Status	Reduce disk and memory	Flexibility
Open5GCore [11]	Commercial	v	Active	v	v
OpenEPC [23]	Commercial	v	Active	v	v
nwEPC [24]	v		Frozen		
openair-cn [25]	v		Active		

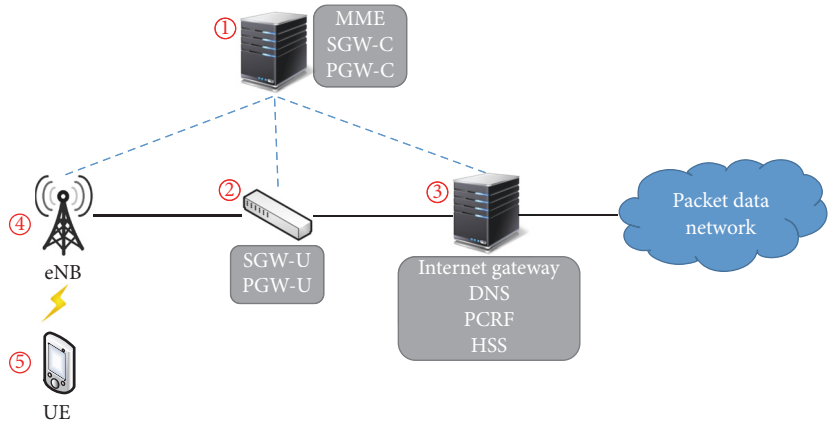


Figure 5: The architecture of Open5GCore.

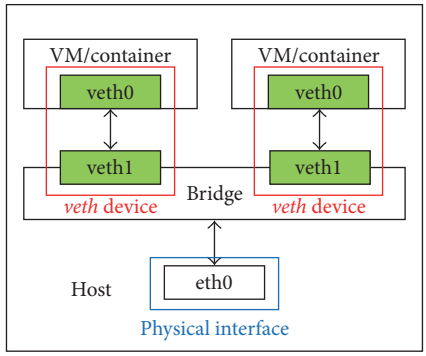


Figure 6: An illustration of the veth network.

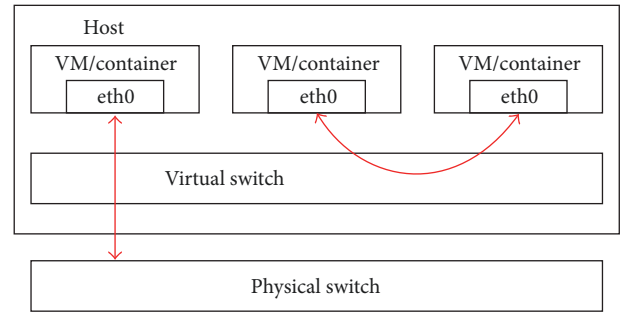


Figure 7: Bridge mode of *macvlan/macvtap*.

an earlier version of Open5GCore. Although Open5GCore is a commercial product, it provides the source code. As far as we know, some universities and research institutes have purchased or plan to purchase Open5GCore to conduct research. It is a software implementation and can run on commodity PCs. The information provided in this paper would be helpful for other researchers who have purchased or will purchase Open5GCore and want to virtualize Open5GCore. Besides, because Open5GCore is a complete implementation following the 3GPP standards R11, the experimental results and analysis shown in this paper would be similar to those with other commercial products which do not provide source code. The results shown in this paper should be valuable to those, such as operators, seeking the results in commercial products.

In addition to the implementation of EPC, Open5GCore also separates the control plane and data plane. Figure 5 shows the architecture of the Open5GCore we purchased. The control planes of SGW (SGW-C) and PGW (PGW-C) are combined with MME into node ①. The user planes of SGW (SGW-U) and PGW (PGW-U) are implemented to node ②. Other functions, such as HSS, DNS, and PCRF, are integrated into node ③. Although physically they reside together, logically they are different entities. Node ④ is eNB,

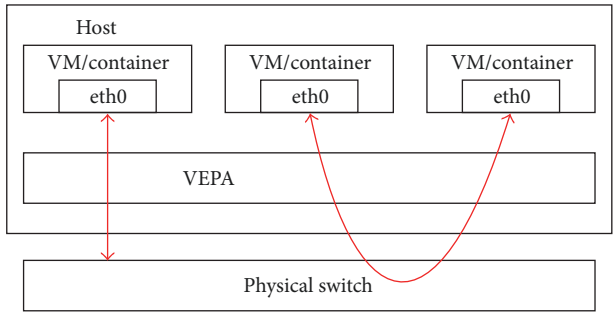


Figure 8: VEPA mode of *macvlan/macvtap*.

which essentially is a base station. Node ⑤ is User Equipment (UE).

Although some functions are combined or divided, all nodes function by following the 3GPP standards. Node ① plays the role of MME, and it is mainly responsible for mobility management. Meanwhile, control planes of SGW and PGW are combined into node ①. Node ② acts as the data planes of SGW and PGW, so node ② also performs policy enforcement and serves as the mobility anchor point for handover between 3GPP and non-3GPP systems. It also performs IP routing and forwarding and runs per-user based packet filtering. Node ③ performs the remaining functions of EPC, such as providing authentication information and user profiles to node ①. Node ④ and node ⑤ act as eNB and UE, respectively.

3. Related Work

Instead of having dedicated hardware for each network, many network systems are considered to be virtualized recently. Eiras et al. virtualized the HTTP proxy over KVM and Docker and compared the performance [9]. Spoiala et al. built the WebRTC over KVM and Docker [10]. Virtualization about cloud computing is evaluated in [29–31]. These articles showed that Docker outperforms KVM in most experiments because Docker is lite.

Network Functions Virtualization (NFV) on carrier-grade networks is also investigated. Tseliou et al. proposed RENEV to virtualize the radio access network (RAN) [32]. The RENEV can create an abstraction of the system's radio resources to serve many base stations. In [33], the authors virtualized MME by dividing it to three components: a front end, works, and a state database. The virtualized MME can effectively reduce the UE attaching latency. Bousia et al. proposed an infrastructure-sharing algorithm that allows operators to share resources and switch off redundant base stations [34]. Li et al. proposed DPCM to eliminate some instructions to enhance the performance of LTE control plane [35].

People are also interested in knowing whether Docker performs well on carrier-grade networks. As far as we know, however, there are very few studies about it. Aman Jain et al. built a simple EPC and evaluated the performance over SDN and NFV. However, they only built three procedures, initial attach, detach, and traffic flow, and did not follow the complete 3GPP standards. Fontenla-Gonzalez et al. installed OpenEPC [23], the predecessor of Open5GCore, on Docker and analyzed the memory usage [36]. However, the authors only analyzed memory usage. Besides, KVM was not included.

To make up for the deficiencies, we installed Open5GCore, a commercial carrier-grade core network implementation, on both KVM and Docker. Furthermore, we analyzed the performance of Open5GCore on physical machines, KVM, and Docker.

Table 2: Specifications of host.

CPU	Intel Core i7 6700
Memory	DDR4 2133 MHz 16 G
Chipset	Intel B150
Storage	SATA3 1 TB
NIC	Gigabit PCI-E

```
/opt/Open5GCore
| -- bin
| -- etc
| -- log
| -- mm_gui
| -- run
| -- sbin
| -- ser_ims
| -- wharf
```

Figure 9: The file structure of Open5GCore.

4. Virtualizing Open5GCore

Open5GCore is a commercial product. We purchased it with source code. In this section, we show the procedures of installing Open5GCore on physical machines and virtual machines. Because KVM and Docker are the most popular ones for hypervisor and container, respectively, we chose them for our virtual machines. The installations of Open5GCore on physical machines, KVM, and Docker are tedious and sometimes troublesome if there is no script to speed up the setup process. The lessons we have learned can save time for others. In this section, we briefly present the installations. We will publish a technical report to describe detailed procedures. In addition to the installations, we will also show the comparison and performance results in Section 5.

We installed Open5GCore on Ubuntu 14.04. The hardware specifications are listed in Table 2. The file system is categorized into (1) binaries, (2) configuration, (3) logging, (4) framework and modules including open source modules, and (5) client/UE as shown in Figure 9. We describe them below.

(1) In Open5GCore, bin/ and sbin/ contain system binaries which can execute, halt, and reboot the system. (2) The system configuration files are gathered to etc/ which includes the configuration of core network entities, configuration of system database, and Open5GCore system reconfiguring scripts. (3) Logging is provided for system information

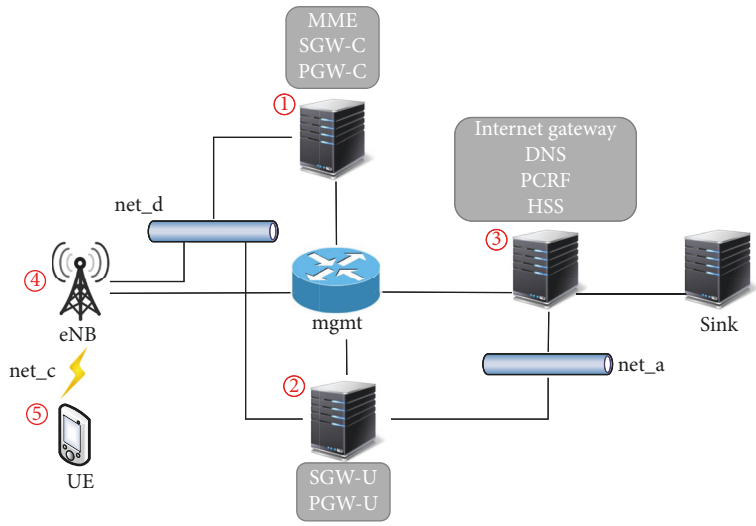


Figure 10: Interface names of Open5GCore.

Table 3: Entity name of each node.

Node	Entity name
Node ①	sgwc-pgwc-mme
Node ②	sgwu-pgwu
Node ③	enablers
Node ④	enodeb
Node ⑤	client

Table 4: Packages required by KVM.

Packages names	Description
qemu-kvm	KVM core engine
libvirt-bin	API and library
ubuntu-vm-builder	Tool for Ubuntu to create VM
bridge-utils	Enable bridge function and configuration on host system

tracking and debugging which are located at run/ and log/. (4) The most important directory is the wharf/ which contains Open5GCore's framework and binaries of modules. Due to the nondisclosure agreement, the details of wharf will not be discussed in this paper. An open source IP Multimedia Subsystem (IMS) for Call Session Control Function (CSCF) is under ser_ims/. (5) Lastly, an emulated UE client written in JAVA is placed under mm_gui/.

Open5GCore uses the *entity name* in the source code to represent nodes in Figure 5 during the installation. Table 3 shows the mapping between the entity name and the node number shown in Figure 5.

The following sections discuss the crucial points to solve the potential issues during installation.

4.1. Open5GCore on Physical Machines. The installation of Open5GCore can be easily done by executing the scripts in the source. There are three main steps: (1) change interface names, (2) set interface IP addresses, and (3) install necessary packages and configure the system.

4.1.1. Change Interface Names. In Open5GCore, it is required to set the network interface names according to the rules set by Open5GCore. Otherwise, in next steps, it will not be possible to set IP addresses to the network interfaces. The interface names are *mgmt*, *net_a*, *net_c*, and *net_d* shown in Figure 10. Changing interface names can be done by executing the script file *make_udev.sh* under /sbin.

4.1.2. Set Interface IP Addresses. Running the file *set_ip.sh* under /etc/network can set IP addresses to the interfaces automatically. Once the IP addresses are set correctly, in next step, Open5GCore will connect to the Internet to download other necessary packages.

4.1.3. Install Necessary Packages and Configure the System. This is the main step of the installation. The file *configure_system.sh* under /etc installs the necessary packages automatically via *apt-get*, configures the system, and runs up the entity. For each entity, you will need to run the script separately by specifying the entity names shown in Table 3 to bring up the entity. Necessary packages include *MySQL* [37], *ethtool* [38], and *Gstreamer0.10* series packages [39].

4.2. Open5GCore on KVM. Generally, KVM creates a VM and lets an OS ride on the virtual machine. The guest OS thinks itself owns the machine. Therefore, once the installation of VM is done, the rest of the procedures are the same as those in Section 4.1. There are four main steps: (1) install KVM, (2) set interfaces for the VM, (3) create a VM, and (4) run the standard installation procedures.

4.2.1. Install KVM. The package *qemu-kvm* for KVM needs to be installed by using the *apt-get* command. Some required packages must also be installed by using *apt-get* command. Some of the required packages are listed in Table 4.

4.2.2. Set Interfaces for the Virtual Machine. To make the network interfaces access the network outside the host, network bridges are required to connect the virtual interfaces in KVM to the physical interfaces of the machine before the virtual machine can run successfully. We set each interface required by Open5GCore as a *macvtap* device over the corresponding physical interface one by one.

4.2.3. Create a Virtual Machine. *virt-manager* is a GUI for KVM and can be installed by using *apt-get* command. By using *virt-manager*, virtual machines can be easily created.

4.2.4. Run the Standard Installation Procedures. Run the steps in Section 4.1 to complete the Open5GCore installation.

4.3. Open5GCore on Docker. Because a container of Docker only holds the application level of a system, the installation of Open5GCore on Docker is more complicated. The steps are as follows: (1) pull a Docker image, (2) create a container, (3) setup the interfaces, (4) manually install necessary packages, (5) tune the system for MySQL, (6) disable optimization, (7) run the standard installation process, and (8) manually start up the entities.

4.3.1. Pull a Docker Image. To create a container, we need an image as the base. The image can be *pulled* from the official Docker site, in which there are different images for different OSs. We used *docker pull Ubuntu:14.04* command to pull the image for Ubuntu 14.04.

4.3.2. Create a Container. Because container is an application level process and system-related configuration cannot be set after a container is running, some steps are needed when executing container run-up command:

(i) Set *system mode* to privileged mode: it is very important to set privileged mode for Docker. It makes it possible to change the configuration of network interfaces, which is needed during Open5GCore installation.

(ii) Set hostname: after installing Open5GCore, it will set the system hostname to the name of each entity shown in Table 3. However, after a container is running, the hostname cannot be changed. Therefore, the hostname should be set in this setup.

(iii) Set *network mode* to none: the network settings cannot be changed in the default network mode of Docker. To make the network configurable, the network mode should be set to *none*.

4.3.3. Setup the Interfaces. The container is now in the status without any network interface in none mode. We need to add network interfaces manually before installing Open5GCore. The process to setup interfaces of a container is tedious if there is no script to speed up the setup process. The main steps are as follows: (1) use *ip link add* command to create *macvtap* interfaces as VEPA mode, (2) check the container process ID (PID) via *docker inspect –format* command, and (3) insert the created interfaces to the container with *ip link set* command and the container PID.

4.3.4. Manually Install Necessary Packages. In Step (3) in Section 4.1, the necessary packages will be installed automatically when installing Open5GCore on physical machines. In Docker, however, some packages cannot be automatically installed and should be installed manually. They are *Gstreamer0.10* series packages [39], *ethtool*, and *subversion*.

4.3.5. Tune the System for MySQL. MySQL in a container will access libraries in the host. Because the container is in privileged mode, the host system will omit this access. Opening the access permission by using *apparmor_parser* command in the host can solve the problem.

4.3.6. Disable Optimization. Open5GCore optimizes data transmission by setting some variables to network interfaces. The settings are in kernel level, which are not supported by Docker. Therefore, it is necessary to disable the optimization. This can be done by removing the command *pre-up /sbin/ethtool -K $IFACE gso off gro off tso off tx off rx off sg off* in the file called *interface.entity*.

4.3.7. Run the Standard Installation Process. Run Step (3) in Section 4.1.

4.3.8. Manually Start Up the Entities. We run each entity by executing the binary of *wharf* under /etc with *entity.xml* file as the input file for wharf. In Open5GCore, wharf is a dynamic framework to bring up each entity. Note that MySQL must be restarted every time when an entity is started. Otherwise, MySQL will encounter connection error.

5. Evaluation

In this section, we aim to evaluate Open5GCore implemented on physical machines, KVM, and Docker. We will present the evaluation results for (1) VoIP, Video, and FTP profiling to show the network performance, (2) CPU and memory usages to show the resource utilization, and (3) entities recovery time to show availability.

5.1. Environment Setup. We installed Open5GCore on Ubuntu 14.04. The hardware specifications are listed in Table 2. We use several tools to measure and evaluate the systems, including *IxChariot* [40] to measure the network performance, *sysstat* [41] to check CPU usage, and *watch* [42] to check memory usage.

We follow the Open5GCore architecture to set up the nodes, including node ① to node ⑤ shown in Figure 10. The UE (node ⑤) attaches to *eNB* (node ④) via *net_c*. Node ①, node ②, and *eNB* connect to each other via *net_d*. Node ② and node ③ are connected via *net_a*. All of the entities are connected via *mgmt* for control plane.

5.2. System Performance. In this section, we present the experimental results for the performance of VoIP, video, and FTP. IxChariot [40] is a professional network profiling tool. It provides several real-case network flow scenarios to profile the network quality. Based on the Open5GCore

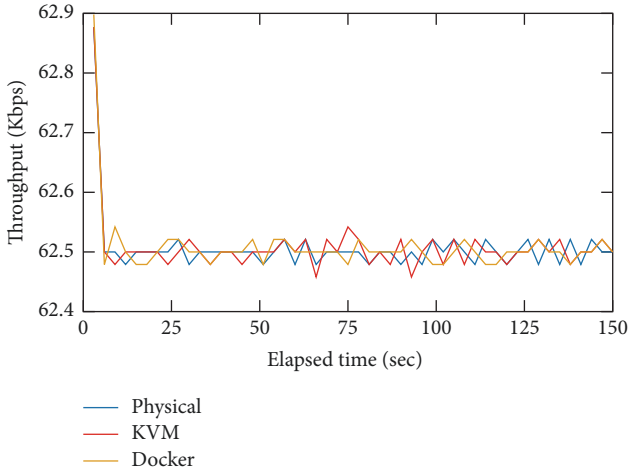


Figure 11: Throughput of VoIP.

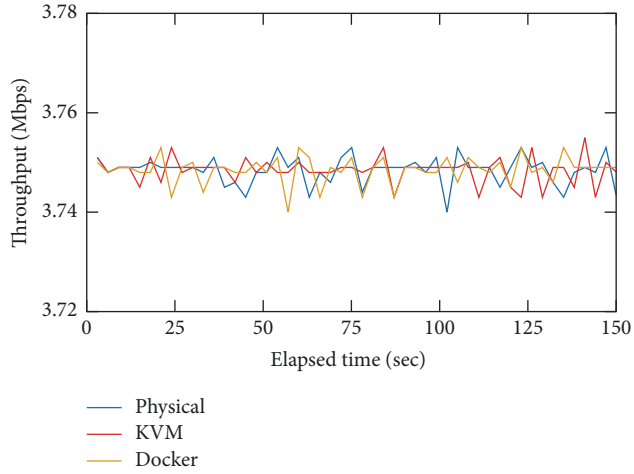


Figure 12: Throughput of video.

architecture, we attached two UEs to node ④ and measured the performance of VoIP, video, and FTP traffic generated by IxChariot. Although FTP is not used very often in mobile devices, it can exhibit the situation of heavy network traffic. Therefore, we include FTP to represent heavy data traffic.

5.2.1. VoIP and Video. In this experiment, we measure the most common use cases in mobile communications: VoIP and video. Both of the VoIP call and video length are 2.5 minutes. Figures 11 and 12 show the throughput of VoIP and video, respectively. The results of physical machines, KVM, and Docker are almost identical and steady, except the high throughput at the beginning of VoIP, which is due to the VoIP hand shaking. In this experiment, we found that, in light traffic, either KVM or Docker can perform as good as physical machines in terms of network throughput.

For delay, on the other hand, Figure 13 shows that Open5GCore on physical machines generally has the lowest delay. This is because the network interfaces in virtual machines (KVM and Docker) suffer extra overhead due to virtualization. Because KVM virtualizes the whole system while Docker only virtualizes the application level, the delay in Docker is little lower than that in KVM.

5.2.2. FTP. As discussed previously, FTP stands for a scenario of large and steady data transmission. Figure 14(a) shows that physical machines perform the best in terms of FTP throughput. The reason is that the overhead in physical machines is the lowest. The difference between physical machines and KVM/Docker can be as large as 3 Mbps. The throughputs of KVM and Docker are close because both of them use the same interface virtualization technique, *macvtap*.

Figure 14(b) shows the results of FTP delay. With the same reason, physical machines have the shortest delay.

5.3. Resource Utilization. Figure 15 shows the results of CPU and memory utilizations when background traffic increases. For CPU utilization, in general, KVM is the highest. In Figures 15(a) and 15(e), KVM incurs high overhead. This is

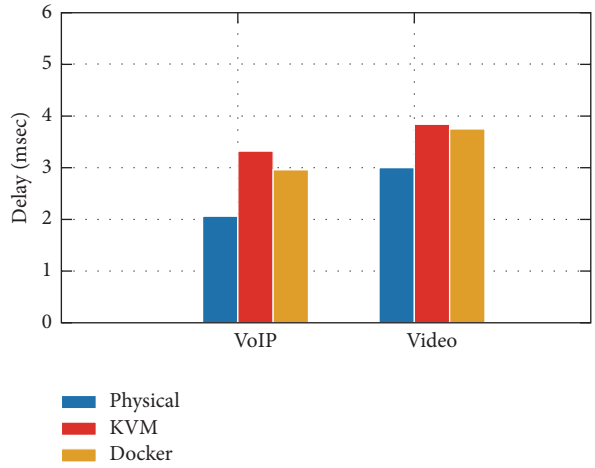
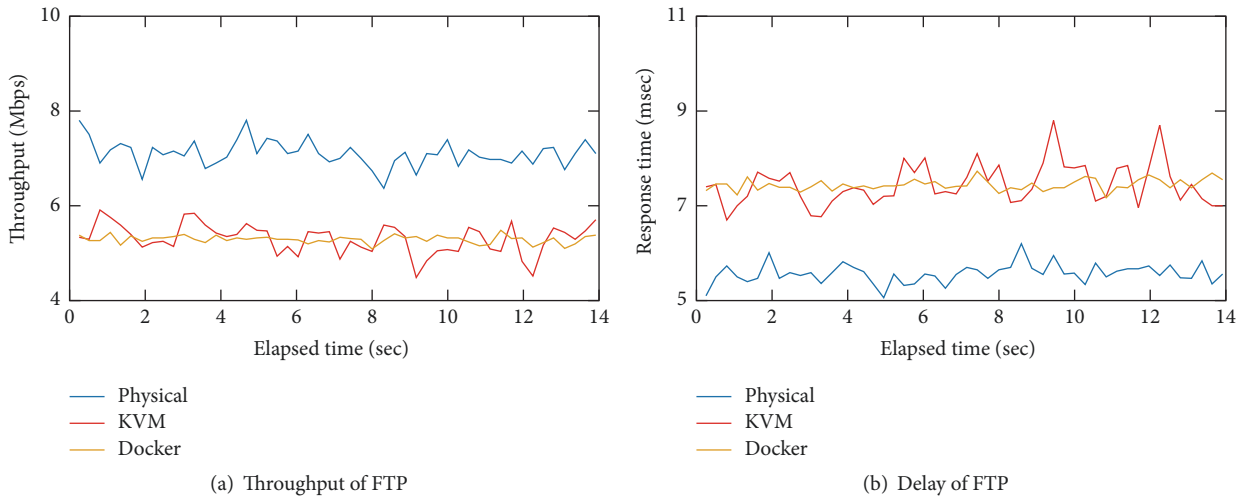


Figure 13: Average one-way delay of VoIP and video.

because KVM needs to run both host OS and guest OSs. Docker, on the other hand, can be seen as an application as shown in Figure 4. Because nowadays the CPU is much fast and memory is big, the overheads of Docker running as an application will not cause many overheads. Therefore, Docker performs almost the same as the physical machine. In Open5GCore, the data plane and control plane are separated. Because node ① is in control plane only, the amount of traffic is less. On the other hand, eNB needs to handle traffic in both data plane and control plane; it consumes much more system resources. For Figure 15(c), all of the results are similar. It is because the background traffic is sent through data plane. Node ① is responsible for control plane. Therefore, KVM in node ① is similar to others.

Figure 15 also shows the results of memory utilization. In general, KVM is the highest and physical machine is the lowest. Once a virtual machine is created, the required memory has been allocated. Increasing background traffic will not increase memory too much.

5.4. Availability. For availability, we refer to system boot time T_{boot}, reboot time T_{reboot}, and recovery time T_{recovery}.



(a) Throughput of FTP (b) Delay of FTP

Figure 14: The performance of FTP.

These are important performance metrics for carrier-grade networks.

"Notations" shown at the end of the paper shows the detailed definition. In Docker, the definition of boot time T_{boot} is the time to start a container in *existed* state. In KVM, it is time to start a VM in *shutdown* state. For NFV in 5G, it is important to know the boot time in Docker or KVM. Reboot time T_{reboot} is the time to restart a running VM/container. In Docker, the recovery time T_{recovery} equals $T_{\text{clone}}+T_{\text{create}}+T_{\text{boot}}$, while the recovery time in KVM is equal to $T_{\text{clone}} + T_{\text{boot}}$. The recovery time refers to the time to bring up a backed up VM/container.

In the experiment, we further define T_{boot_i} as the boot time of the ith container/VM. Assuming that there are n containers/VMs running in the host, we measured the average boot time of n numbers of containers/VMs, which is denoted as $T_{\text{boot\_avg}}(n)$:

$$T_{\text{boot}_{\text{avg}}}(n) = \frac{\sum_{i=1}^{n} T_{\text{boot}_i}}{n}. \tag{1}$$

Similarly, we denote T_{reboot_i} and T_{recovery_i} as the reboot time and recovery time of the ith container/VM, respectively. When there are n containers/VMs, we measured the average reboot time and recovery time of n numbers of containers/VMs, which are denoted as $T_{\text{reboot\_avg}}(n)$ and $T_{\text{recovery\_avg}}(n)$, respectively:

$$\begin{aligned} T_{\text{reboot\_avg}}(n) &= \frac{\sum_{i=1}^{n} T_{\text{reboot}_i}}{n}, \\ T_{\text{recovery}_{\text{avg}}}(n) &= \frac{\sum_{i=1}^{n} T_{\text{recovery}_i}}{n}. \end{aligned} \tag{2}$$

Figures 16 and 17 show the experimental results of $T_{\text{boot\_avg}}$ and $T_{\text{reboot\_avg}}$. Therefore, the time for physical machine is ignored. In the figures, x-axis is the number of VMs/containers. Same as the reasons stated above, KVM has host and guest OSs. Once the number of VMs reaches 12, the system starts to be overloaded. Therefore, the boot time starts to increase as shown in Figure 16(a). Compared with KVM, the boot time of Docker shown in Figure 17(a) is much smaller. In addition, Docker can create much more containers. This is also because Docker is simply an application. For reboot time, Figure 16(b) indicates that KVM can bring up an inactive VM much faster when compared with the boot time shown in Figure 16(a). For Docker, Figure 17(b) shows the same behavior. Because the boot time of Docker is already short, the difference between Figures 17(a) and 17(b) is not that big as the difference between Figures 16(a) and 16(b). Figure 18 plots $T_{\text{recovery\_avg}}$. The discussion is similar to that of boot time because recovery basically is to create a VM/container from a backup. Table 5 shows the average normalized ratios of boot time, reboot time, and recovery time of Docker over KVM.

Table 5: Normalized ratio of Docker and KVM.

		Normalized ratio (Docker/KVM)
Boot time	eNB	0.0525
	Node ①	0.0524
	Node ②	0.0496
Reboot time	eNB	0.1352
	Node ①	0.0940
	Node ②	0.0768
Recovery time	eNB	0.0057
	Node ①	0.0054
	Node ②	0.0056

5.5. Security. Because Docker is in application level, it cannot directly access libraries and the host OS. Thus, it should be quite secure. However, this also precludes the container from setting the system permissions. Thus, Docker cannot build sockets or set up network interfaces. This leads to the fact

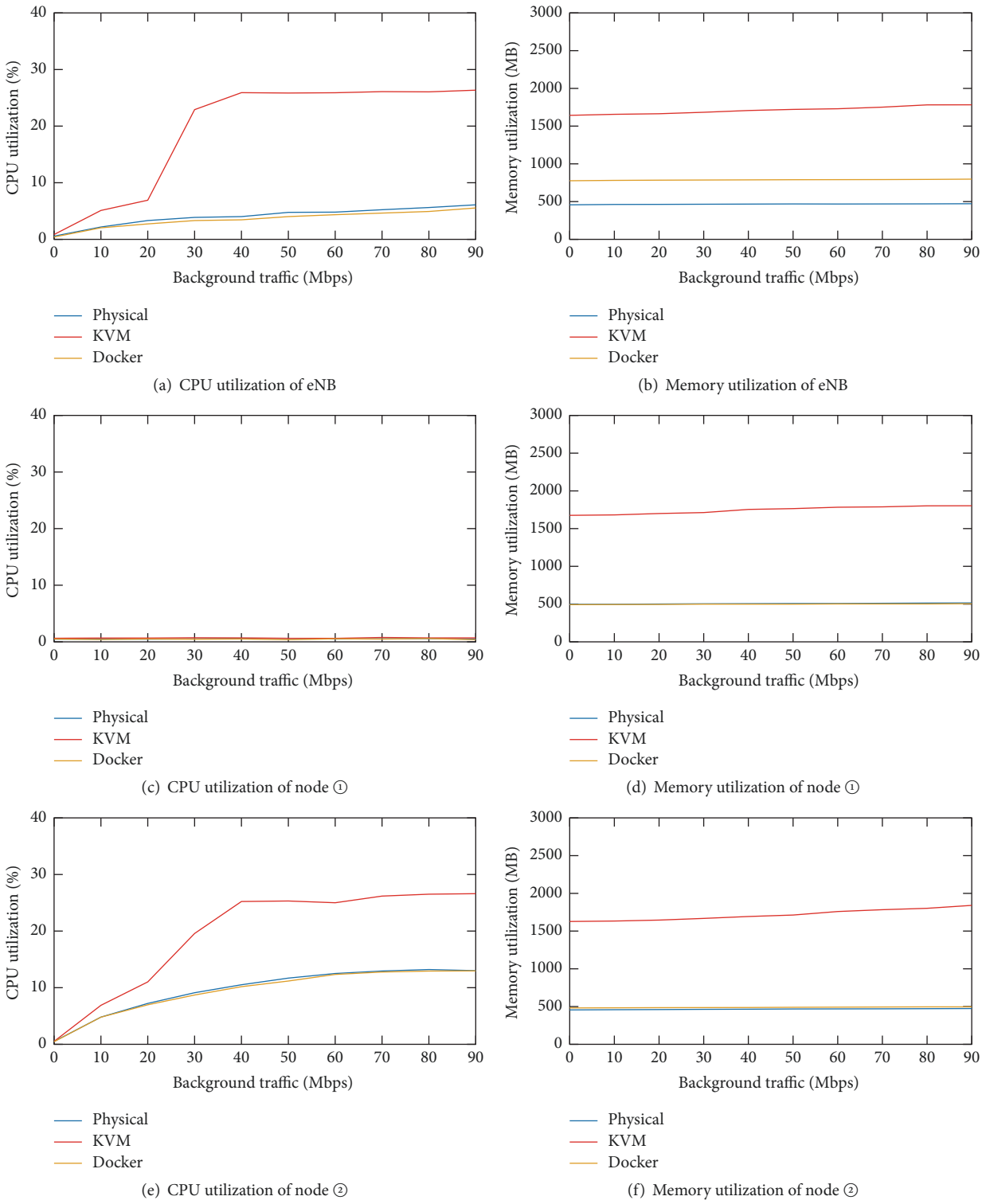
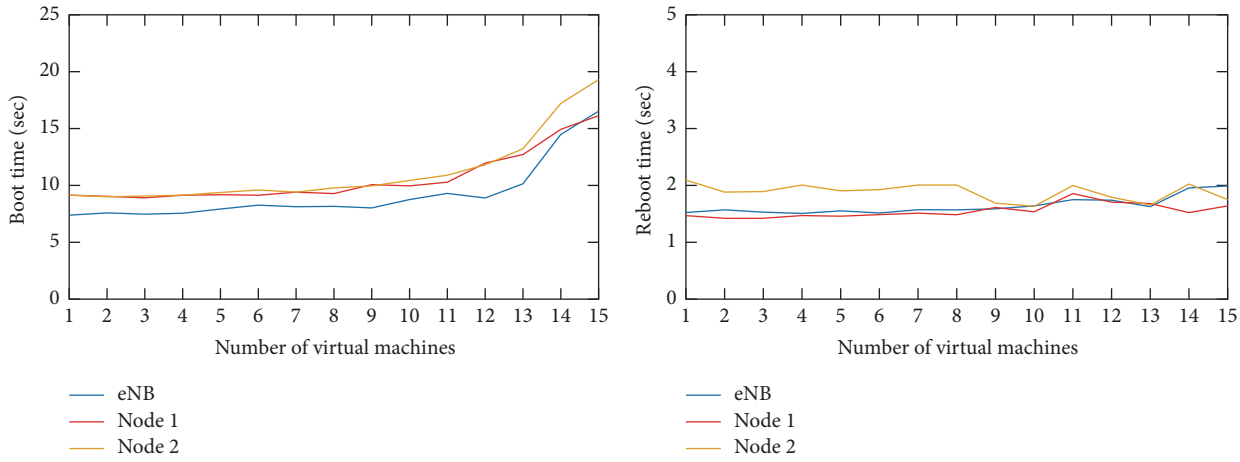


Figure 15: Resource utilization of Open5GCore.

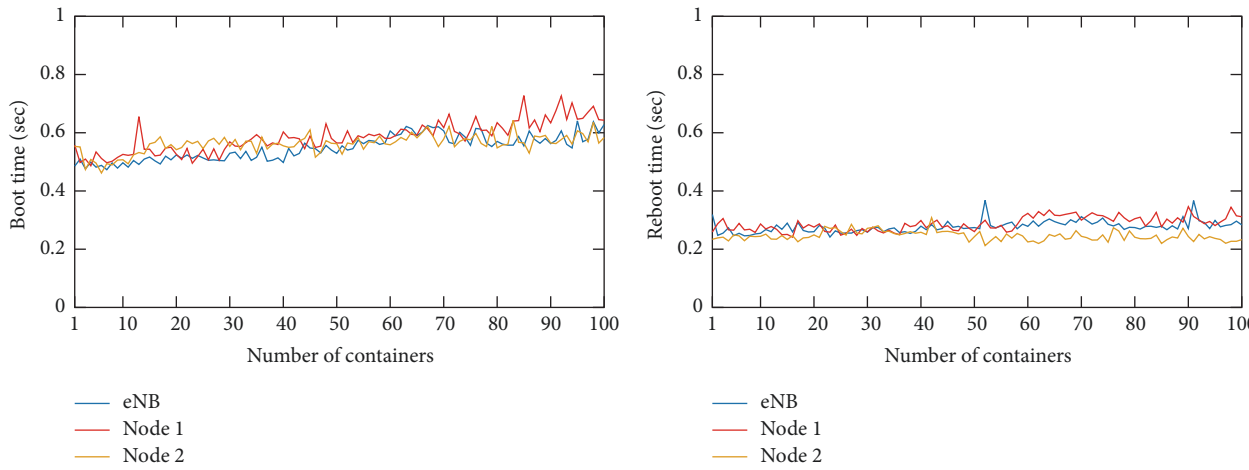
that Open5GCore cannot be installed on Docker successfully. Fortunately, Docker provides privileged mode so that it can grant permissions to set up host configurations. However, this also opens a door for potential security risks.

Even in privileged mode, the *AppArmor* [43] mechanism can restrict users from accessing some protected files or devices. With the protection of AppArmor, unfortunately, MySQL cannot create sockets to access the databases. This



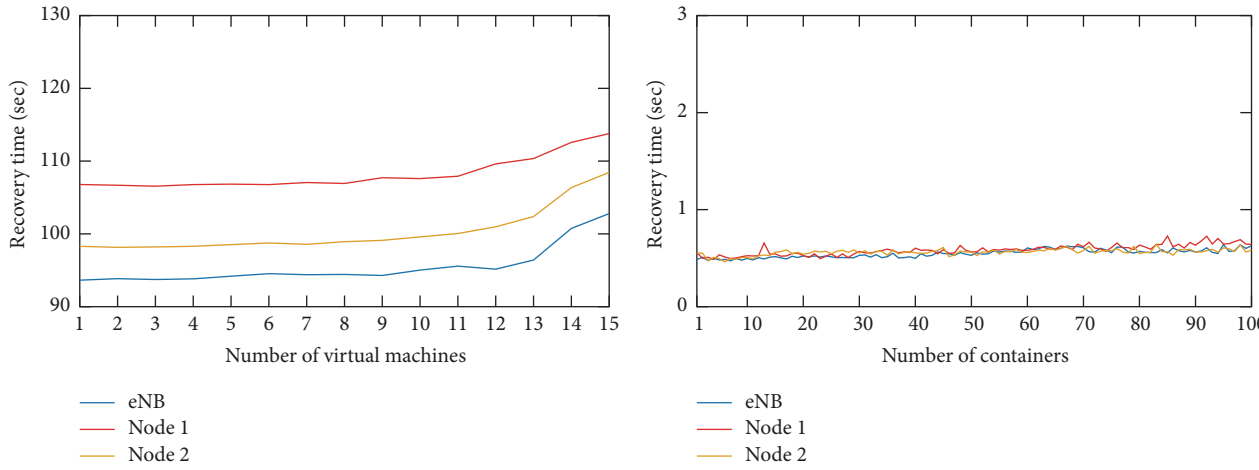
(a) Boot time of KVM (b) Reboot time of KVM

Figure 16: The boot and reboot time of KVM.



(a) Boot time of Docker (b) Reboot time of Docker

Figure 17: The boot and reboot time of Docker.



(a) Recovery time of KVM (b) Recovery time of Docker

Figure 18: Recovery time (KVM and Docker).

Table 6: Operations allowed in privileged mode.

Operation	Risk
Control the devices in the host	Read the host devices and set harmful configuration to the host
Load modules	Compile and load malicious kernel modules
Change ownership of files	Leak confidential data
Access raw sockets	Intercept and manipulate the data (packet spoofing)
Change network settings	Redirect packets to unsafe sites
Modify AppArmor	Alter the number of capabilities of a user

problem can be solved by removing the restrictions of AppArmor on MySQL. However, this leads to another security risk because malicious users can attack the host through the MySQL server.

Traditionally, Linux uses *capabilities* to manage the permissions of a user [44]. A capability allows a user to perform an operation. For example, a user can set the IP address of a network interface if the user has the capability CAPNETADMIN. A container of Docker, by default, only has 14 capabilities [45]. With these capabilities, few operations are allowed, such as making arbitrary changes to file UIDs and GIDs, writing a user ID to map into a user namespace, and using *chroot* environment. Because none of these 14 capabilities is related to system-level operations, a container has high security by default. However, the privileged mode gives a container as many as 37 capabilities. With these 37 capabilities, a container may encounter some security issues. Table 6 lists some operations which can be performed by these 37 capabilities and the associate security risks. (1) The first security issue is that a user can access all devices on the host, including using *mount* command, permitting a user to read unknown IO devices, or setting harmful configurations to host devices that make the host system out of order. (2) A user can compile and load malicious kernel modules to crack the system. (3) A user can change ownership of files to leak confidential data. (4) A user can access raw sockets to intercept and manipulate the data. (5) A user can change network settings, such as changing IP addresses and routing table, to lead network packets to unsafe sites. (6) A user can change the setting of *AppArmor* which can even alter the number of capabilities of a user. This is a serious security issue. We have performed all operations listed in Table 6 successfully. Because the issues listed in Table 6 are operations which can be performed successfully or not, they are not like other metrics which can plot figures. Therefore, we simply discuss them without showing any figures. It is interesting to note that Table 6 also shows the pros/cons of Docker's privileged mode. The operations that can be performed essentially are the pros with the privileged mode. However, those operations also lead to some security risks, which are the cons.

6. Summary and Lessons Learned

In this paper, we present the virtualization of Open5GCore and discuss the performance evaluation. Installing Open5GCore on physical machines, KVM, and Docker is tedious and sometimes troublesome if there is no script to speed up the setup process. Missing one simple step may cause hundreds or thousands of hours to figure out the problem. The lessons we have learned can save time for others.

Based on the performance results in Section 5, we summarize the lessons we have learned:

(i) KVM and Docker will increase extra network delay. If the same interface virtualization technique is used, the delays in KVM and Docker are close.

(ii) KVM/Docker can work as good as the physical machines in light-weight traffic, such as VoIP or video. However, the performance gap becomes large when the traffic is heavy, for example, high-volume data transmission.

(iii) Docker consumes less CPU and memory than KVM does. In addition, Docker can have more instances than KVM.

(iv) The boot time, reboot time, and recovery time of Docker are much less than those of KVM. Besides, Docker can boot, reboot, and recover in a very short time (around 0.5 seconds in our testbed).

(v) Components in control plane may be more suitable for virtualization.

(vi) Virtualization techniques, including KVM and Docker, will incur extra overheads. In general, physical machine still has higher performance.

(vii) Although Docker performs better than KVM, especially on availability, Docker must work in privileged mode to set the network interfaces. Setting it to privileged mode causes security issues.

For future work, we plan to conduct more experiments for more types of traffic. We also plan to test the scenarios with mobility. We also plan to investigate more about the security issues when virtualizing 5G networks. In addition, the overhead of virtualization will be further characterized to help determine what type of virtualization method suits which service. It can serve an EPC NFV orchestrator to better manage resources according to expected delays/performance.

Notations

T_{boot}: Boot time: the time to start a container in *existed* state or a VM in *shutdown* state
T_{save}: Save time: the time to save the state of a running container or a running VM
$T_{restart}$: Restart time: the time to restart a running container or VM
T_{reboot}: Reboot time: $T_{save} + T_{restart}$
T_{clone}: Clone time: the time to clone an image of a container or a VM in the host
T_{create}: Create time: the time to create a container from the image
$T_{recovery}$: Recovery time: Docker: $T_{clone} + T_{create} + T_{boot}$; KVM: $T_{clone} + T_{boot}$.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported in part by the Ministry of Science and Technology (MOST) of Taiwan under Grant nos. 106-2221-E-009-046-MY3, 106-2221-E-009-047-MY3, and 106-2218-E-009-016.

References

[1] A. Osseiran, F. Boccardi, V. Braun et al., "Scenarios for 5G mobile and wireless communications: the vision of the METIS project," *IEEE Communications Magazine*, vol. 52, no. 5, pp. 26–35, 2014.

[2] ETSI, "Network functions virtualisation," 2014, https://portal.etsi.org/Portals/0/TBpages/NFV/Docs/NFV_White_Paper3.pdf.

[3] 3GPP, TS 23.002: Network architecture, 1999.

[4] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal, "NFV: state of the art, challenges, and implementation in next generation mobile networks (vepc)," *IEEE Network*, vol. 28, no. 6, pp. 18–26, 2014.

[5] P. Demestichas, A. Georgakopoulos, D. Karvounas et al., "5G on the Horizon: key challenges for the radio-access network," *IEEE Vehicular Technology Magazine*, vol. 8, no. 3, pp. 47–53, 2013.

[6] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

[7] P. Barham, B. Dragovic, K. Fraser et al., "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pp. 164–177, New York, NY, USA, October 2003.

[8] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *Proceedings of the ACM SIGOPS Operating Systems Review*, vol. 41, pp. 275–287, 2007.

[9] R. S. V. Eiras, R. S. Couto, and M. G. Rubinstein, "Performance evaluation of a virtualized HTTP proxy in KVM and Docker," in *Proceedings of the 7th International Conference on the Network of the Future, NOF '216*, pp. 1–5, 2016.

[10] C. C. Spoiala, A. Calinciuc, C. O. Turcu, and C. Filote, "Performance comparison of a WebRTC server on Docker versus virtual machine," in *Proceedings of the 13th International Conference on Development and Application Systems, DAS '16*, pp. 295–298, 2016.

[11] Open5GCore, 2016, http://www.open5gcore.org/.

[12] Virtualization overview, 2017, https://www.vmware.com/pdf/virtualization.pdf.

[13] A. Khan, A. Zugenmaier, D. Jurca, and W. Kellerer, "Network virtualization: a hypervisor for the internet?" *IEEE Communications Magazine*, vol. 50, no. 1, pp. 136–143, 2012.

[14] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: a survey on concepts, taxonomy and associated security issues," in *Proceedings of the 2nd International Conference on Computer and Network Technology (ICCNT '10)*, pp. 222–226, Bangkok, Thailand, April 2010.

[15] Xen, 2013, https://www.xenproject.org/.

[16] VMWare, 2017, http://www.vmware.com/.

[17] Virtual PC, 2017, http://www.microsoft.com/windows/virtualpc/default.mspx.

[18] VirtualBox, 2017, https://www.virtualbox.org/.

[19] KVM, 2017, http://www.linux-kvm.org/.

[20] QEMU, 2016, http://www.qemu-project.org/.

[21] LXD, 2017, https://www.ubuntu.com/containers/lxd.

[22] Docker, 2017, https://www.docker.com/.

[23] OpenEPC, 2013, http://www.openepc.com/.

[24] nwEPC, 2017, https://sourceforge.net/projects/nwepc/.

[25] openair cn, 2017, https://gitlab.eurecom.fr/oai/openair-cn.

[26] J. Claassen, R. Koning, and P. Grosso, "Linux containers networking: Performance and scalability of kernel modules," in *Proceedings of the 2016 IEEE/IFIP Network Operations and Management Symposium, NOMS '16*, pp. 713–717, April 2016.

[27] F. Fokus, 2017, https://www.fokus.fraunhofer.de/en.

[28] OpenAirInterface, 2017, http://www.openairinterface.org/.

[29] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "KVM, Xen and Docker: A performance analysis for ARM based NFV and cloud computing," in *Proceedings of the 3rd IEEE Workshop on Advances in Information, Electronic and Electrical Engineering, AIEEE '15*, pp. 1–8, November 2015.

[30] B. Xavier, T. Ferreto, and L. Jersak, "Time Provisioning Evaluation of KVM, Docker and Unikernels in a Cloud Platform," in *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid '16*, pp. 277–280, IEEE, Cartagena, Colombia, 2016.

[31] F. Ramalho and A. Neto, "Virtualization at the network edge: a performance comparison," in *Proceedings of the 17th International Symposium on a World of Wireless, Mobile and Multimedia Networks, WoWMoM '16*, pp. 1–6, June 2016.

[32] G. Tseliou, F. Adelantado, and C. Verikoukis, "Scalable RAN Virtualization in Multitenant LTE-A Heterogeneous Networks," *IEEE Transactions on Vehicular Technology*, vol. 65, no. 8, pp. 6651–6664, 2016.

[33] K. A. G. Premsankar and S. Luukkainen, "Design and implementation of a distributed mobility management entity on OpenStack," in *Proceedings of the IEEE Cloud Computing Technology and Science*, 2015.

[34] A. Bousia, E. Kartsakli, A. Antonopoulos, L. Alonso, and C. Verikoukis, "Game-Theoretic infrastructure sharing in multi-operator cellular networks," *IEEE Transactions on Vehicular Technology*, vol. 65, no. 5, pp. 3326–3341, 2016.

[35] Z. Y. Y. Li and C. Peng, "A control-plane perspective on reducing data access latency in LTE networks," in *Proceedings of the ACM Mobile Computing and Networking*, 2017.

[36] J. Fontenla-Gonzalez, C. Perez-Garrido, F. Gil-Castineira, F. J. Gonzalez-Castano, and C. Giraldo-Rodriguez, "Lightweight container-based OpenEPC deployment and its evaluation," in *Proceedings of the 2nd IEEE NetSoft Conference and Workshops (NetSoft)*, pp. 435–440, 2016.

[37] MySQL, 2017, https://www.mysql.com/.

[38] ethtool, 2005, http://linuxcommand.org/man_pages/ethtool8.html.

[39] Gstreamer0.10, 2017, https://gstreamer.freedesktop.org/.

[40] IxChariot, 2017, https://www.ixiacom.com/products/ixchariot.

[41] sysstat, 2017, http://sebastien.godard.pagesperso-orange.fr/.

[42] watch, 2017, https://linux.die.net/man/1/watch.

[43] AppArmor, 2017, https://help.ubuntu.com/lts/serverguide/apparmor.html.

[44] Linux capability, 2017, http://man7.org/linux/man-pages/man7/capabilities.7.html.

[45] Docker security, 2017, http://open.daocloud.io/docker-rong-qi-de-root-an-quan-ma-allen-tan-docker-02/.