An Empirical Case for Container-driven Fine-grained VNF Resource Flexing

Amit Sheoran*, Xiangyu Bu*, Lianjie Cao*, Puneet Sharma†, Sonia Fahmy*
*Purdue University †Hewlett Packard Labs
e-mail: {asheoran, xb, cao62}@purdue.edu, puneet.sharma@hpe.com, fahmy@purdue.edu

***Abstract*—In this paper, we make a case for using lightweight containers for fine-grained resource flexing for Virtualized Network Functions (VNFs) to meet the demands of varying workloads. We quantitatively compare the VNF performance and infrastructure resource usage of three instantiations (bare metal, virtual machine, and container) of three selected VNFs. The three VNFs we experiment with are the Mobility Management Entity (MME) of the Evolved packet core (EPC) architecture for cellular networks, the Suricata multi-threaded Intrusion Detection System (IDS), and the Snort single-threaded IDS. Our results show that container-based instantiations not only incur lower resource usage but also have shorter boot time. This makes containers an attractive choice for fine-grained VNF resource flexing. The lessons learned from our empirical case studies with EPC and IDS provide important guidelines for building an elastic micro-service architecture for NFV deployments.**

I. Introduction

Communication Service Providers (CSPs) are increasingly adopting Network Functions Virtualization (NFV) in their infrastructure. Two primary factors driving NFV adoption are efficient resource usage, and agility in terms of elastic resource allocation [1]. However, CSPs face key challenges in this NFV transformation [2]. While virtualization allows service and resource allocation agility, virtualization of network functions needs to be implemented with minimal overhead for efficient resource usage. *Virtual machines* (VMs) and *containers* are the two most widely deployed virtualization mechanisms in the cloud. Containers such as LXC [3] and Docker [4] are becoming popular for tenant and application isolation in cloud ecosystems. Compared to VMs, containers exhibit lower overhead and higher performance.

In this paper, we compare the performance and resource usage of three Virtualized Network Functions (VNFs) with bare metal (BM), container, and Virtual Machine (VM) instantiations, at a variety of load levels and resource allocation configurations. The three VNFs we benchmark are: (1) The Mobility Management Entity (MME) of the Evolved packet core (EPC) architecture for cellular networks, (2) the Suricata multi-threaded Intrusion Detection System (IDS), and (3) the Snort single-threaded IDS. To the best of our knowledge, ours is the first extensive empirical study that compares resource usage efficiency and elastic resource flexing of VMs and containers for VNF deployment.

We demonstrate that container-based deployments incur significantly lower overhead than VM-based deployments, and reduce the system initialization time. This makes containers an ideal choice for systems where booting delays directly affect how quickly additional resources can be allocated to VNFs (agility), and higher elasticity is desired to reduce the overall operational cost. We experiment with deploying multiple instances on the same hardware platform, and find that such an elastic deployment model provides high resource utilization without the need for re-architecting sub-optimal VNF implementations such as single-threaded applications.

Thus, the contribution of this paper is threefold:

1) We benchmark the performance of the Mobility Management Entity, Suricata, and Snort VNFs on bare metal, containers and VMs under varying workloads and with different numbers of instances.
2) We analyze the time required by the VNFs to start on VMs and containers.
3) Based on our results, we make several recommendations for resource flexing of VNFs, including VNFs based on legacy single-threaded software.

II. Virtual Network Functions

We consider two types of VNFs for our empirical study: (1) an evolved packet core (EPC) control plane VNF, MME, and (2) two IDS VNFs (Snort and Suricata).

The selection of these VNFs is based upon the following criteria: (1) Plane of operation: An MME is primarily a control plane element. Typical control plane elements are CPU-intensive and do not stress the data plane of the network. An IDS, in contrast, is a data plane entity that monitors all incoming packets in the network and thus is network-intensive. (2) Software architecture: Snort is single-threaded, and can only utilize a single CPU core. MME and Suricata are multi-threaded systems. Due to implementation choices, MME only saturates two CPU cores at peak capacity whereas Suricata can scale to utilize all available CPU cores. (3) Network positioning: MME is a stateful transactional element that communicates with other EPC elements to handle user requests. IDSes are typically deployed as stand-alone middleboxes, and do not depend on message exchange with other elements. The results allow us to gain insight into the impact of virtualization techniques on VNFs with different system design and requirements.

A. EPC Control Plane

We give a brief introduction to the EPC network and its role in managing the User equipment (UEs) [5]. The basic

978-1-5090-0933-6/16/$31.00 © 2016 IEEE

This work has been sponsored in part by NSF grant CNS-1319924.

LTE architecture is shown in Fig. 1. An LTE network consists of a Radio access network (RAN) and Evolved packet core (EPC). The RAN comprises the eNodeB, which provides radio connectivity to the users. The EPC network consists of the MME, Home Subscriber Server (HSS), Serving Gateway (S-GW) and Packet Data Network Gateway (P-GW). The MME and HSS are control plane entities responsible for signaling, mobility, and security functions for UEs attaching over the RAN. S-GW and P-GW are data path entities responsible for data transfer to and from the UEs.

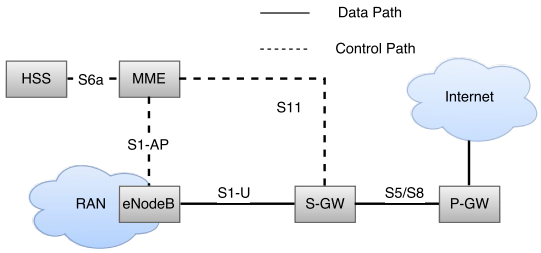

Fig. 1: LTE network architecture [6].

B. Intrusion Detection Systems

Suricata http://suricata-ids.org/ and Snort https://www.snort.org are popular intrusion detection systems. The reason we use both is that Snort is single-threaded whereas Suricata is multi-threaded. This difference allows us to evaluate the impact of the implementation on total system throughput when NFV orchestration systems scale-out by creating additional instances of the same VNF.

III. Experimental Setup

We conduct our experiments on a server cluster in which each server has the following hardware configuration: **CPU** Intel Xeon X3430 @ 2.40 GHz; Nehalem; EIST disabled; HT not supported, **RAM** 2 x 2GB DDR3-1333 and **NIC** 2 x Broadcom 1 Gbps.

In each experiment, a sender host runs a traffic generator and a receiver host runs our VNF. The traffic generator runs directly on the hardware while the receiver uses one of the following three setups: (1) **Bare metal (BM):** The MME/IDS run on the native hardware and read incoming data directly from the NIC. (2) **Virtual Machine (VM):** The VNF is deployed in a guest VM created using QEMU/KVM [7] and a Linux bridge is used to connect the VMs to external hosts. We select KVM as a representative hypervisor for our experiments as KVM-based VMs incur lower overhead while handling CPU and network intensive workloads [8] compared to other popular hypervisors like Xen. (3) **Container (Docker):** The VNF runs in a Docker container and a Linux bridge is used for external connectivity. Unless otherwise noted, the memory limit is set to 2 GB for VMs and containers.

The VM and Docker setups closely resemble their bare metal counterparts. They run the same version of operating system, compiler and VNFs and use the same configuration files and rule sets. Our experiments also include setups where multiple instances of the same VNF are deployed within one physical host; in such cases, the hardware resources are proportionally divided among the VNF instances and a Linux bridge is used to direct traffic to the correct instance.

In the EPC experiments, the MME, S-GW and P-GW nodes use one of the setups discussed above, whereas the HSS and eNodeB emulator are hosted on two physical machines connected via a switch (Fig. 2). We use Ubuntu 14.04.4 64-bit LTS (kernel version **3.19.3-031903-generic**) with the following software: **gcc** 4.8.4, **docker** 1.11.0, **libvirt** 1.2.2, and **qemu** 2.0.0.

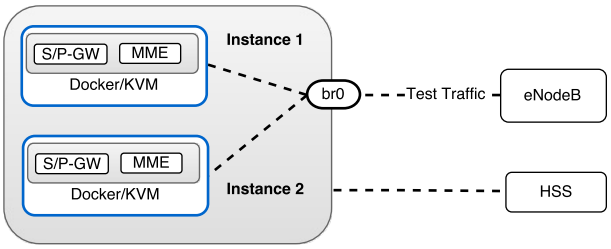

Fig. 2: Setup for EPC tests.

In the IDS experiments, the sender generates test traffic by replaying trace files using multiple instances of TCPreplay (Fig. 3). The IDS receiver host runs Ubuntu 15.10 64-bit with the following software: **gcc** 5.2.1, **docker** 1.11.0, **libvirt** 1.2.16, and **qemu** 2.3. The sender host uses **tcpreplay** 4.1.1 to generate load. We use **suricata** 3.0.1 or **snort** 2.9.8.2 with **EmergingThreat Rules** 20160414. Unless otherwise noted, all 4 CPU cores are accessible, and VT-d is enabled.

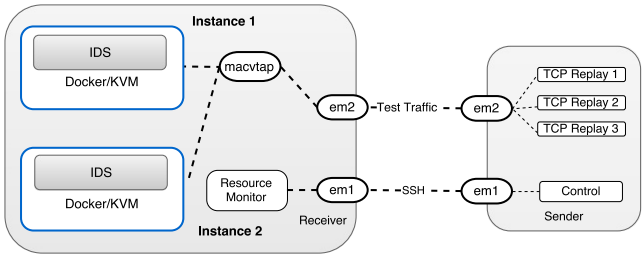

Fig. 3: Setup for IDS tests.

In addition to the setups described earlier, the IDS experiments study performance with the following virtualized networking technologies: (1) **Docker with host NIC:** In this case, the host NIC is directly (exposed to the IDS process in the container instances. (2) **macvtap:** Instead of exposing the host NICs to containers, we create a macvtap device to direct traffic to the IDS process. We use "DockerV" to refer to this setup. We experimented with different means of forwarding traffic, including bridging and Virtual Ethernet Port Aggregator (VEPA), and found that macvtap of mode "passthrough" and model "virtio" incurs the lowest overhead. This is consistent with the findings of Anderson *et al* [9] for macvlan.

IV. EPC Experiments

A. Methodology

We use openair-cn [10] to benchmark the EPC control plane. Openair-cn is a 3GPP-compliant implementation of EPC components that can be executed on general purpose hardware. Our test setup includes an MME node connected to the HSS and a client emulator framework based on the "openairinterface

oaisim" [11] application. The MME is co-located with the S-GW, and all communication between them is handled via internal queues. The client emulator connects to the MME over the S1 interface, and generates attach requests emulating multiple UEs at a constant rate per second.

The eNodeB emulator is used to generate registration requests which result in the exchange of several control plane messages between eNodeB, MME, HSS and the S-GW. During the experiments, we measure the time taken by the MME to successfully process UE registration requests. The experiments are run multiple times to obtain at least 20 samples, and the average time over all samples is used.

The emulator generates registration requests at a constant rate of 400 registration requests per second and the total time taken by MME to respond to a total of 4000, 6000 and 8000 registration requests is measured. We measure the performance of the system with 1, 2 and 4 CPU cores enabled. Since the MME implementation handles most of the processing in a single thread, enabling multiple cores allows us to analyze the scale-out capability of the VM and container deployments.

B. Results

We analyze the time taken by each setup to handle the registration requests. The results in Fig. 4 show that VMs incur significantly higher overhead than the bare metal setup and Docker containers. The VM can result in 10-22% overhead, whereas the overhead of Docker is 0-3%. The traffic generation rate for these experiments is 400 registrations/second. It takes 10, 15 and 20 seconds to generate the required 4000, 6000 and 8000 requests, respectively. We note that time taken by the MME to handle these sessions increases as the number of concurrent sessions handled increases. More sessions timeout when the system load is higher.

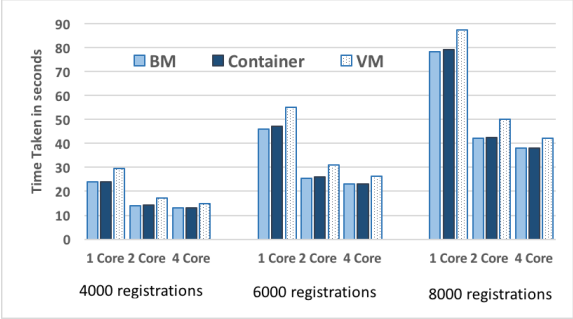

Fig. 4: Time taken to handle registration requests.

Since MME is a transaction-based system and stores all the active sessions in memory, the maintenance overhead increases with number of active sessions. This includes the indexing time required to fetch and store session-related data in the internal data structures, and timers maintained to handle events like timeout and heartbeats. The current implementation of MME uses a hash-based indexing mechanism to store the UE information. The likelihood of collisions and chaining increases when the number of active sessions being handled by an instance increases.

As noted earlier, the MME is co-located with the S-GW and P-GW. Consequently, when only a single CPU core is available, performance of the system is constrained by the available processing power. When two cores are available to the system, performance of the MME significantly increases because the processing thread uses one of the CPU cores, and other features can use the second core. However, this performance benefit is not observed when four cores are available, confirming that the system is single-threaded. This implementation limits the ability of the MME to utilize all available cores in the system as only a single instance of the application can be instantiated.

C. MME Performance Evaluation with Scale-out

A key advantages of virtualization is the ability to deploy multiple instances of the VNF on the same physical machine. While VMs emulate virtual hardware on the guest OS using hypervisors, containers leverage operating system-level isolation using Linux kernel features like cgroups and namespace. Having determined the inability of the MME to utilize all available CPU cores, we now investigate deploying multiple MME instances on the same host. This technique, commonly referred to as *scale-out*, is used in data center environments and enables CSPs to exercise fine-grained resource flexing to meet requirements on-demand.

In this test setup, we use two MME instances that share available resources. Both instances are configured at the eNodeB, and the emulator sends the request to each instance in a round-robin fashion. We generate the same number of registration requests as the earlier setup, but the number of registration requests handled by each instance is halved. Consequently, the number of active sessions handled by each instance is reduced to 2000, 3000 and 4000. We also double the request generation rate from the client to 800 registrations/second, so that each MME instance receives the requests at 400 registrations/second. To establish a baseline, we first show the results when a single MME instance handles traffic at the rate of 800 registrations/second on a bare metal machine. These results are presented in Fig. 5. We find that increasing the traffic generation rate does not have a significant impact on performance. This is because the bottleneck when using a single instance of the system is the MME application thread and not the transport receive thread.

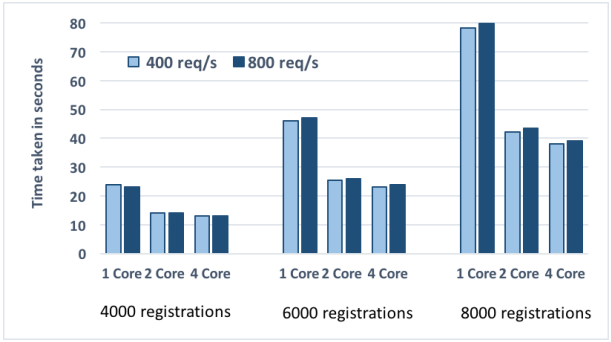

Fig. 5: Time taken to handle registration requests when varying the request rate.

Fig. 6 shows that the time taken to handle the registration requests is considerably reduced when traffic is split across two instances with similar processing resources. Additionally,

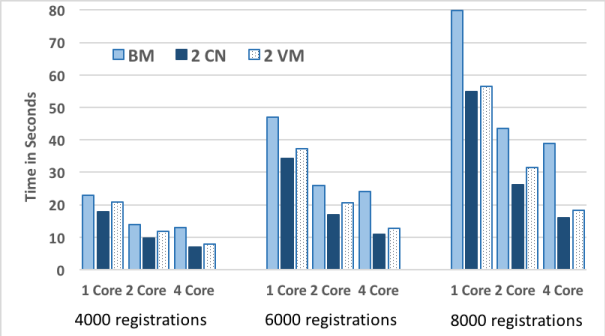

Fig. 6: Time taken to handle registration requests by bare metal and two instances of VMs or containers.

as the number of CPU cores increase from 2 to 4, we find that the time taken by the MME to handle registration requests decreases which indicates better utilization of available processing resources, compared to the case when only a single MME instance was used. While these results reflect the behavior of our MME implementation and may not be directly applicable to other commercial deployments, they can be used to infer the benefits of a micro-services architecture in transaction-based systems.

D. Instantiation Time

In cases when on-demand virtual instances are created to meet network demand, it should be possible to provision a new instance of the MME without incurring significant delay. The MME boot time is the time taken to initialize internal data structures, create transport connections and establish the diameter application-level connection.

TABLE I: Comparison of activation time.

Metric	Bare Metal	Docker	VM
Time(s)	4.77	4.81	12.01

As shown in Table I, the time taken by the Docker container to start is much closer to the time taken by bare metal, whereas the time taken by the VM is considerably higher, due to the overhead involved in loading and booting the guest-OS kernel and hypervisor.

V. Suricata Experiments

A. Methodology

We benchmark Suricata based on statistics it reports and resource usage of the entire host machine. The reason why we examine resource usage of the entire system is that both Docker and QEMU have overhead not reported by their APIs. For example, Docker's `stat` API only reports resource usage inside the container and excludes Docker itself. The CPU and RAM overhead of forwarding traffic is not included in the Suricata, Docker, or QEMU processes. Therefore, comparing resource usage of the entire system is more comprehensive. **Trace Files:** We use two trace files: (1) `bigFlows.pcap` provided by TCPreplay. According to the TCPreplay site, it captures "real network traffic on a busy private network's access point to the Internet" and contains 40686 flows and 132 network protocols [12]. It sends 359,457 KB of data in 791,615 packets in 5 minutes, and (2) `snort.log.1425823194` – one of the ISTS '12 trace files [13]. It generates 155,823 KB of data in 22 seconds. **Increasing Load:** To increase load, more TCPreplay processes may run in parallel. We will use "load level" to denote how many concurrent processes are used. For example, "4X load" means that a test has four TCPreplay processes concurrently replaying the trace file. **Aggregating the Results:** We run each test configuration at least 30 times and take the median of the samples to generate a representation of the test. Before generating a sample, the receiver host is rebooted to restore system state back to the original. We measure host memory and CPU usage, and the number of packets captured, analyzed (decoded), and dropped by the IDS. We do not examine the number of alerts triggered because it is highly affected by the packet drops.

B. Results

We first analyze resource usage by comparing memory and CPU utilization of the Suricata host running different setups at various load levels, then compare the performance of Suricata in different setups.

1) Memory Usage: Our results indicate that the memory overhead of Docker is trivial compared to bare metal, whereas the VM setup consumes substantially more memory. Table II shows the memory usage (average and standard deviation σ) of Suricata in VM, Docker and bare metal setups at 1X workload. Docker has a small memory footprint since the host and Docker container have shared libraries of the same version, eliminating the need to load more libraries. Although the VM runs the same software setup, a full-fledged guest operating system must be maintained, which results in high memory overhead.

TABLE II: Memory usage of Suricata at 1X workload.

Metric	Bare Metal	Docker	DockerV	VM
Average	9.85	10.20	10.22	23.83
σ	0.33	0.33	0.34	2.29

We also tested Docker, DockerV, and VM with 4X `bigFlows.pcap` and a smaller 512 MB memory limit. While the first two worked without problems, memory thrashing occurred with the Suricata VM before CPU became the bottleneck.

2) CPU Usage: Table III shows the CPU usage of Suricata with Docker, VM and bare metal with 1X workload. Docker does not impose significant CPU overhead, while the CPU usage of the VM setup is considerably higher.

TABLE III: CPU usage of Suricata at 1X workload.

Metric	Bare Metal	Docker	DockerV	VM
Average	19.65	20.92	21.81	269.39
σ	4.61	4.66	4.90	49.66

Comparing Table III to Table IV, we find that the CPU usage of the bare metal, Docker and DockerV setups multiplies corresponding to load level, but the VM setup saturates the CPU at 2X workload. This results in the CPU becoming a bottleneck at workloads greater than 2X. We also observe that

TABLE IV: CPU usage of Suricata at 2X workload.

Metric	Bare Metal	Docker	DockerV	VM
Average	40.75	42.11	44.00	399.90
σ	9.87	9.90	10.44	0.70

the Docker setup incurs an overhead of 1% to 4% depending upon the load level. There is a roughly 0% to 5% increase in CPU usage associated with macvtap depending on traffic throughput.

3) Packets Received: At all four load levels we use, packet capture is about the same, but the VM setup tends to receive fewer packets. After the TCPreplay ends, we send a SIGTERM signal to Suricata and wait for it to exit gracefully. In the VM setup, there are likely packets yet to capture when exiting, resulting in the discrepancy observed.

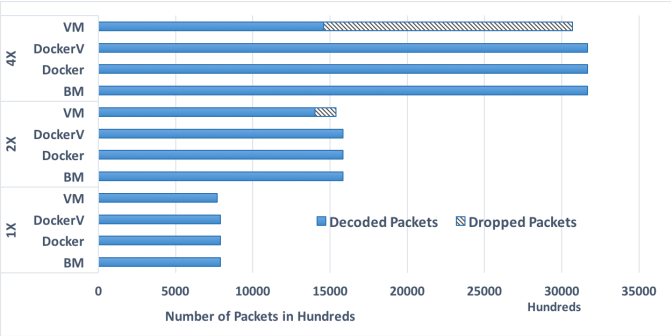

Fig. 7: Cumulative packets decoded and dropped by Suricata in four setups at all loads.

4) Packets Dropped: Dropping packets is a sign that Suricata cannot process the workload with the resource constraints imposed. As seen from Fig. 7, only the VM setup exhibits packet drop starting at 2X load, and almost all increased load above 2X is dropped. This result is consistent with the CPU usage in Table IV. CPU is nearly saturated by Suricata at 2X load level and therefore almost all additional traffic generated at 4X load is dropped.

5) Packets Decoded: From Fig. 7, we find that the Docker and DockerV setups are on par with bare metal in terms of rate of packet analysis (decoding). However, only at 1X load (where CPU has not become a bottleneck), is the rate of packet decoding in the VM setup comparable to bare metal, and the rate decreases as load increases, which reveals severe performance degradation.

C. Results with the Higher Rate Trace

The trace file `snort.log.1425823194` requires the receiver host to use more CPU to handle the high-throughput traffic. Similar results were observed, but we saw bare metal saturated at 4X load. At 4X load, Suricata drops 67,126 packets with bare metal, 67,064 packets with Docker, 81,282 packets with Docker with macvtap, and 282,453 packets with VM. In fact, Suricata on VM runs so slowly that even the kernel drops 205,095 packets to reclaim buffer space before Suricata is able to read them. This confirms that (1) Docker has comparable performance with bare metal, (2) the overhead of macvtap can be nontrivial, and (3) the VM setup is the slowest.

D. Root Cause Analysis for VM Results

To determine the reason for the observed performance degradation, we profiled Suricata running on bare metal and on VM. The culprit we found is the frequently called function 'UtilCpuGetTicks()' which flushes the instruction pipeline and reads the x86 Timestamp Counter (TSC). The x86 instruction 'rdtsc' may cause VM exit [14], and by checking the msr register we verified that it indeed causes VM exit in our VM setup. This makes the instruction expensive in the VM, and results in a massive performance penalty. To further isolate the cause, we ran this sole function 100 million times on bare metal and on VM, and it takes 6.6 seconds (on average) to finish on bare metal, but 126.1 seconds on VM.

VI. Snort Experiments

For the Snort experiments, we use the same setup as Suricata except that Suricata is replaced by Snort. Again, we ran each test configuration at least 30 times, and used the median of each metric. We discuss the outcome at 4X load with `bigFlows.pcap`.

Snort exports statistics only on exit. Although a stop signal was sent to the Snort process 20 seconds after TCPreplay finished, Snort did not stop immediately after receiving the signal.

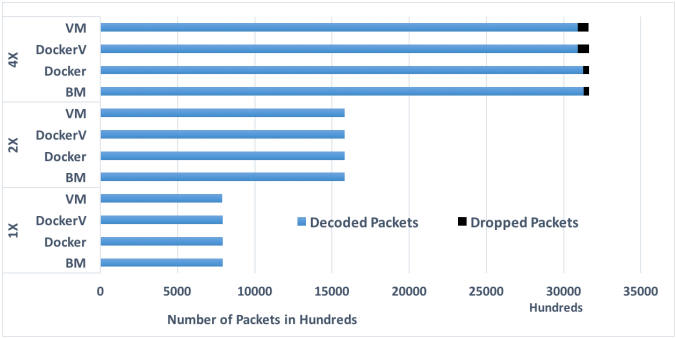

Fig. 8: Cumulative packets decoded and dropped by Snort in four setups at all loads.

Fig. 8 confirms that Snort works best on bare metal, followed by Docker. Surprisingly, DockerV and VM give similar results. There are two factors to take into account: (1) Snort ran slightly longer after receiving the exit signal, and (2) the total number of packets to receive is supposed to be 3,166,460, *i.e.*, as with Suricata on VM at 4X load, Snort on VM failed to capture all packets. Although Snort is single-threaded, the VM overhead caused the VM setup to use more than 100% CPU combined (Table V).

TABLE V: CPU usage of Snort at 4X workload.

Metric	Bare Metal	Docker	DockerV	VM
Average	73.62	75.17	79.42	137.23
σ	13.28	13.80	14.02	18.48

In terms of memory usage, the VM setup uses more than twice the memory of the Docker setup. The Docker and DockerV setups use slightly more RAM ($\sim$1%, or $\sim$40 MB) than that of bare metal (Table VI).

TABLE VI: Memory usage of Snort at 4X workload.

Metric	Bare Metal	Docker	DockerV	VM
Average	12.84	13.25	13.26	27.45
σ	0.40	0.32	0.33	0.36

VII. Suricata and Snort with Multiple Instances

We now investigate multiple instances of Suricata or Snort that are deployed on the same host. Based on our findings from previous experiments, we know that Snort does not effectively utilize all available CPU cores due to its single-threaded design. While this design limits its scalability on bare metal, we can deploy multiple instances of containers and VMs to utilize available CPU cores more effectively.

Since both Snort and Suricata bare metal setups handle the 4X workload without significant packet drops, we use 4X and 8X workloads to evaluate scalability with multiple instances. The values presented in this section represent the median value of at least five runs of each experiment.

In Fig. 9 and 10, CN indicates a Docker setup with a Linux bridge, and VM indicates a VM setup with a Linux bridge. In case of multiple instances of containers or VMs, the value presented is the sum of the number of packets processed by each instance independently: 2-CN and 4-CN indicate the number of packets processed by two and four container instances, respectively, and 2-VM indicates the number of packets processed by the two VM instances collectively.

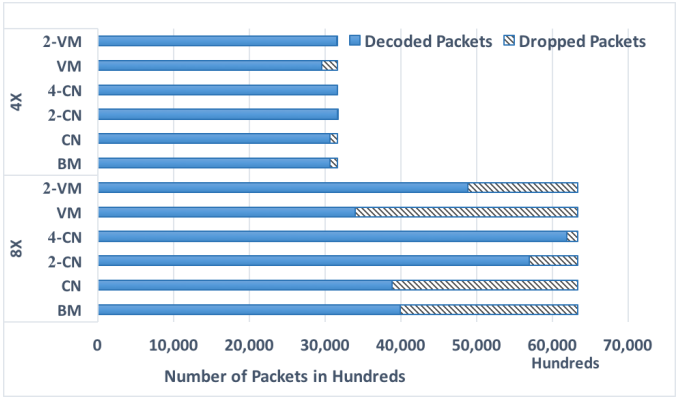

Fig. 9: Performance of Snort with multiple VM and container instances.

Fig. 9 shows that the performance of Snort significantly increases when the incoming traffic is split among multiple instances. The performance difference is more pronounced at 8X workload as a single Snort instance is unable to handle the incoming traffic. We find that the number of packets dropped significantly decreases when two container instances are deployed and continues to decrease with four instances. This behavior is consistent with our earlier finding when multiple instances of MME were deployed on the same host. Further, we note that while two VM instances provide significantly higher performance compared to a single instance of VM and bare metal, the performance benefit is not on par with a multiple container deployment.

As noted above, Suricata is multi-threaded and is capable of utilizing all available CPU cores even with a single instance deployment. Table IV had shown that Suricata saturates the available CPU cores at 2X traffic when deployed as a single instance. From Fig. 10, we find that there is no observable performance difference between the single and multiple instance deployments. While we observe some performance gain (∼1%) in the VM setup when two VM instances are deployed, we find that the factor limiting system performance is the available CPU which cannot be circumvented by deploying multiple instances.

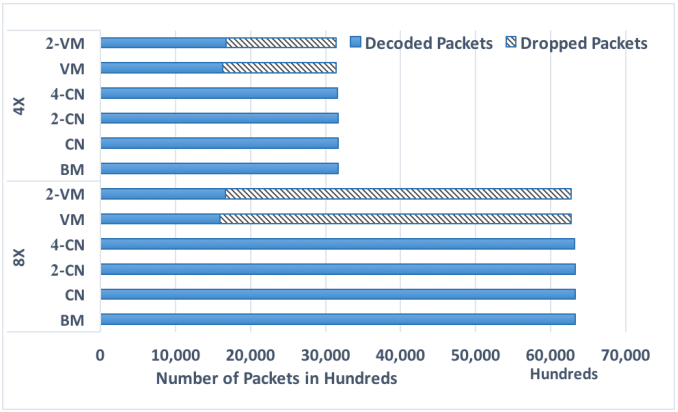

Fig. 10: Performance of Suricata with multiple VM and container instances.

The observations from the EPC and Snort experiments validate the efficacy of a container-based micro-service architecture for legacy software that is constrained from achieving scalability on modern hardware platforms.

VIII. Performance Impact of VT-d and VFIO

Virtualization techniques like Intel's Virtualization Technology for Directed I/O (VT-d) and VFIO passthrough aim to bridge the performance gap between bare metal and VMs via a hardware assist to virtualized software [15]. In this section, we study the impact of these technologies on the performance of the MME and IDSes.

Fig. 11 shows the time taken by the MME to handle registration requests with VT-d and VFIO enabled. VT-d indicates the setup when VT-d is enabled, and PS indicates the setup when both VT-d and VFIO passthrough are enabled. In case of PS, one NIC is dedicated to the MME VM instance. We see that VT-d results in up to 4-7% performance gain. Furthermore, enabling both VT-d and VFIO passthrough yields a performance gain of 5-12%, compared to a basic VM setup.

IX. Related Work

Network Function Virtualization [1] has gained significant momentum over the past few years. Gember *et al* [16] investigated application deployment in the cloud from various perspectives, including elasticity, network flow distribution, and virtual machine placement. The use of NFV in the domains we study (EPC and IDS) has also been investigated. Banerjee *et al* [17] discussed how NFV can help scale EPC systems. Elasticity in intrusion detection systems has been investigated in [18], [19]. None of the above studies specifically compares containers to virtual machines.

More recently, the use of containers has been studied by Anderson *et al* [9] and Kamarainen *et al* [20]. Anderson *et*

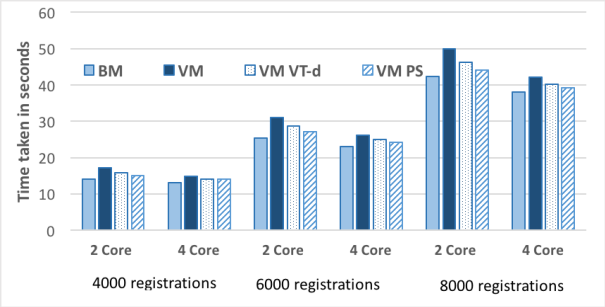

Fig. 11: Time taken by MME to handle registration requests with bare metal and VM with VT-d and VFIO passthrough.

al [9] examined the impact of network technologies like ovs, bridging and macvlan on the throughput of Docker containers. While we use some of these techniques in our experiments, our work focuses on EPC and IDS functions. Kamarainen *et al* [20] explored the impact of virtualization techniques on cloud gaming systems. Their study examines the impact of virtualization on video encoding and hardware resource sharing and does not consider transaction-based systems and network-layer entities as we do.

X. Case and Considerations for Container VNFs

Our results validate that NFV deployments can benefit significantly from the use of containers, similar to the widely adopted use of containers for cloud workloads. Nevertheless, VNF developers and CSP administrators must be careful with performance isolation and available support for containers in NFV software ecosystems.

[Resource usage efficiency] Containerized instantiations of both control-plane (MME) as well as data-plane (Snort and Suricata) VNFs consumed less CPU and memory resources compared to traditional VM-based deployments. MME container instances had 3% CPU overhead over bare metal compared to the 22% overhead incurred by VM instances. Similar CPU usage efficiency was observed in case of IDS VNFs. This is because, with VMs, a full-fledged guest operating system introduces significant memory overhead.

[Resource flexing granularity] The smaller resource usage footprint of containers allows finer granularity resource flexing. There is still, however, a non-zero instantiation overhead for each instance. Fine-grained resource flexing may lead to sudden VNF overload in the event of flash workload bursts. Though VMs incur high instantiation overhead, they provide additional slack to tackle sudden workload bursts.

[Decomposition and micro-services] The smaller footprint of containers makes them an ideal choice for decomposing monolithic VNF implementations. As we showed in our experiments, different threads of a VNF implementation may get loaded differently depending on the offered workload. VNF developers can leverage containers for creating micro-services that can be scaled-out individually to meet varying demands. High VM instantiation overheads make them an unlikely choice for micro-services.

[Poor implementations and lack of thread support] Our experiments highlighted the need for well-architected and well-implemented VNFs. For instance, significant performance degradation was observed in the VM instantiation of Suricata 3.0.1. VNF developers should carefully profile and compare performance of their routines (*e.g.*, 'UtilCpuGetTicks()' in Suricata) on bare metal, containers, or VMs. We have demonstrated a container-based design for creating multiple instances of poorly threaded (single-threaded or monolithic) VNF implementations. Such multi-instance deployments can significantly improve the overall system capacity and performance. This was particularly evident in case of IDS VNFs that need high data-plane throughput.

References

[1] "ETSI Network Functions Virtualisation (NFV) Architectural Framework," http://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_NFV002v010201p.pdf.

[2] L. Cao, P. Sharma, S. Fahmy, and V. Saxena, "NFV-VITAL: A framework for characterizing the performance of virtual network functions," in *Proceedings of IEEE NFV-SDN*, 2015, p. 7.

[3] "Linux containers: Lxc," https://linuxcontainers.org/.

[4] "Docker: An open platform for distributed applications for developers and sysadmins," https://www.docker.com/.

[5] "3rd generation partnership project, general packet radio service (gprs) enhancements for evolved universal terrestrial radio access network (e-utran) access," http://www.3gpp.org/ftp/Specs/html-info/23401.htm.

[6] M. Olsson, S. Sultana, S. Rommer, L. Frid, and C. Mulligan, *SAE and the Evolved Packet Core: Driving the Mobile Broadband Revolution*. Academic Press, 2009.

[7] "Kernel virtual machine," http://www.linux-kvm.org/page/Main_Page/.

[8] J. Hwang, S. Zeng, F. y. Wu, and T. Wood, "A component-based performance comparison of four hypervisors," in *IFIP/IEEE IM 2013*.

[9] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon, "Performance considerations of network functions virtualization using containers," in *Proc. of ICNC*, 2016.

[10] "openair-cn: An implementation of the evolved packet core network," https://gitlab.eurecom.fr/oai/openair-cn/.

[11] "Openairinterface," https://gitlab.eurecom.fr/oai/openairinterface5g/.

[12] "Sample captures - tcpreplay," http://tcpreplay.appneta.com/wiki/captures.html.

[13] "Pcap files from the the information security talent search (ists)," http://www.netresec.com/?page=ISTS.

[14] Intel, "Intel 64 and ia-32 architectures software developer's manual, volume 3b: System programming guide, part 2," pp. 21–13, 2011.

[15] "Enabling intel virtualization technologyfeatures and benefits," http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-enabling-intel-virtualization-technology-features-and-benefits-paper.pdf.

[16] A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella, "Stratos: Virtual middleboxes as first-class entities," University of Wisconsin-Madison, Tech. Rep., 2012, technical report TR1771.

[17] A. Banerjee, R. Mahindra, K. Sundaresan, S. Kasera, J. Van der Merwe, and S. Rangarajan, "Scaling the LTE control-plane for future mobile access," in *Proceedings of CoNEXT*, December 2015.

[18] P. K. Shanmugam, N. D. Subramanyam, J. Breen, C. Roach, and J. V. der Merwe, "DEIDtect: Towards distributed elastic intrusion detection," in *Proceedings of DCC*, 2014.

[19] V. Heorhiadi, M. K. Reiter, and V. Sekar, "New opportunities for load balancing in network-wide intrusion detection systems," in *Proceedings of CoNEXT*, 2012.

[20] T. Kamarainen, Y. Shan, M. Siekkinen, and A. Yla-Jaaski, "Virtual machines vs. containers in cloud gaming systems," in *Proc. of NetGames*, Dec 2015, pp. 1–6.