vNetRadar: Lightweight and Network-Wide Traffic Measurement in Virtual Networks

Tie Ma, Jin Zhang, Long Luo, Hongfang Yu, Gang Sun, Jian Sun
University of Electronic Science and Technology of China, Chengdu, China
mt@std.uestc.edu.cn, {zhangjin, llong, yuhf, gangsun, sj}@uestc.edu.cn

Abstract—**Measuring traffic metrics is indispensable in virtual networks as it is the basis for a wide range of applications, such as network diagnostics and performance evaluation of the network algorithms. However, existing measurement schemes fail to have all these excellent characteristics simultaneously: 1) fine-grained, i.e. to obtain per packet level information. 2) lightweight, namely low CPU and bandwidth overhead. 3) network-wide, which means obtaining metrics of the whole network, e.g. per packet path. 4) easy-to-deploy, which refers to deployment without additional modification of Maximum Transmission Units (MTUs). We design *vNetRadar*, a virtual network measurement system, which has these excellent characteristics simultaneously. Specifically, vNetRadar 1) identifies each packet without increasing the size of each packet, to obtain network-wide metrics without MTU modification, 2) allocates each packet an area in memory, called *backpack*, and carries metadata in it to largely reduce bandwidth overhead. vNetRadar is implemented based on the extended Berkeley Packet Filter (eBPF) and is mainly in kernel space, avoiding the CPU overhead of copying packets to user space when performing the fine-grained measurement. Evaluation results show that the easy-to-deploy vNetRadar can get fine-grained network-wide metrics with low CPU and bandwidth overhead.**

I. Introduction

In recent years, with the development of Network Function Virtualization (NFV), virtual networks have gained wide application in areas such as digital twin networks [3] and network emulation [1] [2]. For example, the well-known mininet [1] is a virtual network emulator and can generate a virtual network with several nodes on a server. In this case, measuring the metrics of the traffic in the virtual network is crucial as it can help diagnose the state of network traffic, verify the performance of network algorithms, etc.

To provide a better view of the virtual network, an ideal network measurement scheme should have the following four excellent characteristics: **1) fine-grained**, i.e. to obtain per-packet level metrics; **2) lightweight**, namely low overhead in terms of CPU and bandwidth. **3) network-wide**, which means obtaining some metrics of the whole network rather than a single measurement point, such as the path of a specific packet. **4) easy-to-deploy**: a scheme should be deployed without MTU modification. Some schemes need to increase the packet size during the measurement, causing the MTUs of the VNFs to need to be manually and carefully modified. Moreover, not every VNF supports the modified MTU natively.

However, existing schemes fail to have these four excellent characteristics at the same time due to the following two challenging conflicts. **1) The conflict between fine-grained and lightweight, caused by carrying metadata in packets and the packet copying.** For instance, the recently emerging In-band Network Telemetry (INT) [8] inserts metadata to packets at each hop, then uses the network to transfer the metadata to a remote metadata analyzer, which occupies a huge amount of network bandwidth. The metadata is the information about the state of the packet as it passes through each hop, including ingress and egress timestamp, etc. tcpdump [12] is a commonly used mirroring measurement scheme that mirrors packets from the kernel data path. The packet copying from the kernel space to the user space results in a large CPU overhead when performing the fine-grained measurement. **2) The conflict between network-wide and easy-to-deploy, caused by the packet identification problem.** To obtain network-wide metrics, it is necessary to identify the same packet at different locations in the network, to correlate the experience of the same packet at different measurement points. Since there are not enough free bits in the packet, a common practice is to increase the packet size and carry an identifier in the extended bits, resulting in the MTU modification and increasing the deployment difficulty. For example, INT's method of carrying metadata within packets implicitly enables packet identification and thus able to obtain network-wide metrics, but pays the price of increasing the packet size and modifying the MTUs. tcpdump [12] does not natively implement packet identification, so it is not network-wide but is easy-to-deploy.

We notice a feature of virtual networks that none of the existing solutions exploit: virtual networks usually build on the same server, thus sharing the same server environment (e.g. CPU, memory, and disk). Leveraging this feature, we would like to propose a measurement scheme that can resolve all the above-mentioned conflicts and thus have all the four excellent characteristics simultaneously. First, to solve the packet identification problem, we want to assign a global unique ID to each packet, whereas the distributed network nodes can only generate local unique IDs. Fortunately, the shared server environment enables us to have a global view, which could use to allocate global unique IDs to packets. To perform packet identification without increasing the packet size, we wish to find some native bits of the packet to carry this

This work is supported by the National Key Research and Development Program of China (2019YFB1802800, 2021YFB3101000) and the National Natural Science Foundation of China (62102066). This work is also being supported by Open Research Projects of Zhejiang Lab (NO. 2022QA0AB02). Corresponding author: Hongfang Yu.

TABLE I: Representative Network Measurement Schemes

Category	Name	Fine-grained	Lightweight: Low CPU Overhead	Lightweight: Low Bandwidth Overhead	Network-wide	Easy-to-deploy
In-band Telemetry	INT [8]	✓	✗	✗	✓	✗
	PINT [9]	✗	✓	✓	✓	✗
	LightGuardian [10]	✓	✓	✓	✓	✗
	PPTMon [11]	✓	✓	✗	✓	✗
Traffic Mirroring	tcpdump [12]	✓	✗	✓	✗	✓
	Everflow [13]	✗	✓	✓	✓	✓
Shared Environment-based Telemetry	**vNetRadar (Ours)**	✓	✓	✓	✓	✓

global unique ID while ensuring these bits function properly. Besides, to solve the bandwidth overhead of carrying metadata inside the packets, we find that although the bandwidth in the virtual network is precious, the server has a large amount of memory. Thus we consider allocating a corresponding area in the server memory for each packet to store metadata, which avoids the network bandwidth overhead of carrying metadata in the packets. Finally, to avoid the packets copying, we prefer to implement the measurement scheme in the kernel space.

In this paper, we present vNetRadar, a fine-grained, lightweight, network-wide, and easy-to-deploy virtual network traffic measurement system. vNetRadar overwrites some bits of the IP header to carry a Global Unique ID (GUID) to uniquely identify individual packets in the virtual network. We maintain the overwritten bits to ensure they work properly, and they are recovered when the packets leave the measurement domain. vNetRadar allocates an area in memory called *backpack* for each in-flight packet[1] and carries metadata in it. Finally, vNetRadar is implemented in the kernel space based on the extended Berkeley Packet Filter (eBPF)[2] to perform packet overwriting, information extraction, and other operations directly on the kernel data path.

The main contributions of this paper are as follows:

- **We design vNetRadar and implement it based on eBPF**. vNetRadar is a lightweight and network-wide traffic measurement system for virtual networks that can be easily deployed and can obtain fine-grained metrics.
- **We design a solution for identifying packets without increasing their size**, which allows network-wide metrics to be obtained without MTU modification.
- **We design the *backpack* mechanism to store metadata**, which largely reduces bandwidth consumption.
- **We demonstrate the application of vNetRadar and evaluate its performance**. Extensive evaluation indicates that the easy-to-deploy vNetRadar can obtain fine-grained network-wide metrics with low CPU and bandwidth overhead, while the operations on the data path introduce a negligible network delay.

II. Background and Related Work

Since we use Extended Berkeley Packet Filter (eBPF) to implement our work (vNetRadar), we begin with a brief introduction to eBPF to better explain this work and then introduce the related work most relevant to this work.

[1]In-flight packets are packets which are sent but have not arrived yet.
[2]"eBPF," https://ebpf.io/

A. Background on Extended Berkeley Packet Filter

Extended Berkeley Packet Filter (eBPF) is a high performance tracing framework inside the Linux kernel. eBPF allows custom programs to be attached to points that will be triggered by the execution of system events or functions, these points are called hooks. eBPF programs can store data in the key-value data structure located in kernel space, which is called BPF map and can be shared between different eBPF programs.

B. Related Work

As shown in TABLE I, based on how the measurement information is obtained, we divide the related network measurement schemes into two categories: **1) In-band Telemetry**. In-band telemetry is the scheme that uses the business traffic to carry metadata. INT [8] is the representative of this type of scheme, it inserts metadata into packets at each hop thus having a large bandwidth overhead. PINT [9] reduces the bandwidth overhead of INT by sampling, which makes it fail to get fine-grained information. LightGuardian [10] achieves measurement by embedding a small field in packet headers, but it grows packet size and requires MTU modification. PPTMon [11] is a scheme based on eBPF, it focuses on the packet latency. Like INT, PPTMon inserts metadata to the packet per hop to get the network-wide latency, but the inserted metadata consumes lots of bandwidth and needs MTU modification. **2) Traffic Mirroring**. Traffic mirroring schemes export traffic from the data plane to get a view of the network. tcpdump [12] is a famous traffic mirroring scheme with a huge CPU overhead. Everflow [13] selectively matches and mirrors specific packets to be measured, decreasing the CPU overhead but leading to coarse-grained. In summary, to the best of our knowledge, none of the existing network measurement schemes except our shared environment-based scheme, vNetRadar, has these excellent characteristics simultaneously: fine-grained, lightweight, network-wide, and easy-to-deploy.

III. Design and Implementation of vNetRadar

In this section, we first introduce the architecture and workflow of vNetRadar. Then we show the two key ideas: the packet identification solution and the *backpack* mechanism. Finally, we discuss the metrics supported by vNetRadar.

A. vNetRadar Architecture and Workflow

As shown in Fig. 1, the virtual network uses the Virtual Network Functions (VNFs) as network nodes, in vNetRadar architecture, there are three types of VNF roles that collaborate

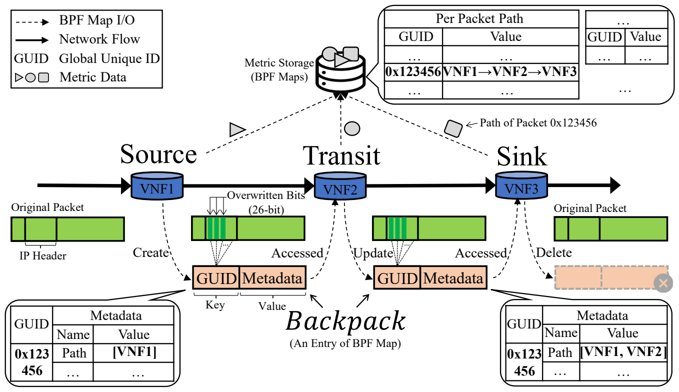

Fig. 1: vNetRadar architecture and workflow. In this figure, we demonstrate how vNetRadar measures the path of a packet.

to obtain the metrics of each packet as it passes by: *source*, *transit*, and *sink*.

The *source* is the first hop of the measurement path[3]. When a packet arrives at a *source*, the *source* overwrites some bits of its IP header to allocate it a GUID. This way, vNetRadar is able to uniquely identify this packet in the virtual network. Then the *source* creates a key-value entry in a shared BPF map, called *backpack*, to carry metadata and the original value of the overwritten bits in it. To guarantee one packet uniquely corresponds to one *backpack*, we use the GUID of the packet as the key of the *backpack*. The *backpack* can be accessed by the *transit* and the *sink* later.

The *transit* is the middle hop. Using the GUID in the IP header of the packet as the key, the *transit* gets the *backpack* corresponding to the packet. The metadata in the *backpack* can be used to calculate some network-wide metrics of the packet such as path and per-hop latency. The metadata is updated by the *transit* using its local information (e.g. current timestamp and device ID). Besides, since the overwritten bits of the IP header may be changed by the VNFs, the *transit* is responsible for maintaining these bits to keep the GUID remaining the same when the packet reaches the next hop.

The last hop, *sink*, also can access the *backpack* to calculate the network-wide metrics. Specifically, the *sink* restores the overwritten bits using their original value in *backpack* to guarantee transparency outside the measurement path. Finally, to prevent the *backpacks* from occupying memory for a long time, the *sink* deletes the *backpack* in the BPF map.

On demand, the *source*, *transit*, and *sink* can store metrics in vNetRadar's metric storage module, which is implemented using several BPF maps. Since a VNF can be the first hop and the last hop on the measurement path of different measured flows simultaneously, a VNF can act as a *source* and a *sink* at the same time. The VNF roles are implemented by our eBPF program which is attached to the Traffic Control (TC) *clsact* [14] hook at egress. This hook is located at the bottom of the kernel network stack, and our program process each packet as it is about to leave the VNF. Since our eBPF program is an

[3]The measurement path of a packet means its path in a vNetRadar-enabled network.

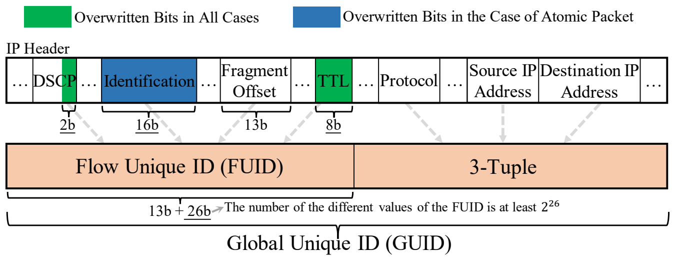

Fig. 2: Packet identification solution. We overwrite some bits of the IP header and combine them with several other fields to form the Global Unique ID (GUID).

additional procedure on the data path and does not disrupt the original function of VNF, vNetRadar supports multiple kinds of VNF such as router, firewall, etc.

B. Packet Identification Solution

Why is packet identification required? To get the network-wide metrics, individual packets should be consistently traced across the network, which requires us to identify each packet. For example, to obtain the path of a specific packet, we need to identify this packet among the large number of packets received by each VNF.

Why not use the IP ID field to uniquely identify each packet? To identify each in-flight packet in the network, each in-flight packet should have a unique identifier, i.e. the number of different values of the identifier should be greater than the number of in-flight packets, which equals the bandwidth-latency product of traffic plus the number of packets in bottleneck buffer [6]. A natural idea is to use the IP ID field as the identifier. According to the RFC 6864 [4], the IP packets are divided into the atomic packets and the non-atomic packets, the former are packets that are not and will not be fragmented, and the latter are packets that are already fragmented or may be fragmented in the future. For the atomic packets, the IP ID is meaningless [4], and some traffic sources may generate non-varying IP IDs, which prevents the IP ID field used to identify the packets; for the non-atomic packets, the IP IDs are unique but only have 65536 different values, which are not enough even in a small buffer scenario (e.g. a 5 MB buffer can hold 78125 64-byte packets) or a small bandwidth-latency product scenario (e.g. a 1 Mpps flow on a 1 second latency link has 10^6 in-flight packets).

Our packet identification solution. We design the GUIDs to identify individual in-flight packets in the virtual network. The GUID is assigned by *source* and maintained by *transit*. As shown in Fig. 2, instead of other schemes that increase the packet size to perform the packet identification, we directly use or overwrite some native fields of the IP header to carry the GUID. The GUID includes two parts, the 3-tuple (source IP address, destination IP address, and protocol) and the Flow Unique ID (FUID). We use the primitive function of the 3-tuple to identify each IP flow. The FUID consists of the Differentiated Services Code Point (DSCP) field, the Identification (ID) field, the Time To Live (TTL) field, and the Fragment Offset field of the IP header. For the DSCP field, we

TABLE II: Contents of the Currently Implemented Metadata

Name	Description
ttl_original	The original TTL value in the packet
dscp_original	The original DSCP value in the packet
ip_id_original	The original IP ID value in the packet
first_hop_ts	ns level timestamp at the *source*
previous_hop_ts	ns level timestamp at the previous hop
device_id	Device ID
path	An array which represents the packet path

overwrite its least important 2 bits, which has little impact on the Quality of Service (QoS) policy [7]. For the Identification field, since it is unique if the packet is non-atomic, we only overwrite it in the case of atomic packets. For the TTL field, we overwrite all 8 bits of it. To ensure the TTL field remains the same at each hop and can perform the function of the TTL correctly, as the packet passes through the VNF (e.g. a router) which decreases the value of the TTL field, we increase the value of the TTL field in the packet by 1 and decrease the original TTL value in $backpack$ by 1. Once the original TTL value of the packet becomes 1, we set the value of the TTL field to 1 to make the packet be dropped by the VNF. For the Fragment Offset field, we directly use it to identify non-atomic packets which have the same 3-tuple and IP ID.

Why does our packet identification solution work? Here we discuss if the number of the GUIDs is enough to identify each in-flight packet. On one hand, the GUID consists of the 3-tuple and the FUID, but in the worst case, there is only 1 IP flow, thus we only discuss the number of the FUIDs. The Fragment Offset field of the FUID only has specific values when the packet is non-atomic and is always 0 when the packet is atomic, besides, the number of the different values of the overwritten bits of DSCP, ID and TTL field is 2^{26} (i.e. $6.7*10^7$), thus the number of FUIDs is at least 2^{26}. On the other hand, the number of the in-flight packets can be divided into two parts: the number of the packets in the buffer and the bandwidth-latency product. In a large buffer scenario, taking a 50 MB buffer as an example, it can only hold 0.78M 64-byte packets, which is small compared to the number of the FUIDs. For the bandwidth-latency product, we regard the maximum bandwidth as 2.5 Mpps, which is the forwarding rate of DPDK vSwitch, the fastest forwarding tool in the virtual networks [5]. Even if the traffic reaches the maximum rate, the GUID is insufficient when the packet latency reaches 26.8 s ($2^{26}/2.5*10^6$), which is a large value and is hard to reach. In summary, the number of the GUIDs is greater than the number of the in-flight packets, thus our identification solution works.

C. $Backpack$ Mechanism

To obtain the network-wide metrics, it is necessary to store metadata for each packet like INT. Since carrying metadata inside the packet is expensive and consumes lots of bandwidth, we design the $backpack$ mechanism to carry the metadata in the server memory.

Data structure. A $backpack$ is an entry of a BPF map. The key of the $backpack$ is the GUID, which ensures that one packet uniquely corresponds to one $backpack$. The value is the metadata, which is used for packet recovery and network-wide metrics calculation. In our current implementation, the content of the metadata in $backpack$ is shown in TABLE II.

Maintenance of the number of the *backpacks*. To achieve control over memory usage, we maintain the number of *backpacks* approximately equal to the number of in-flight packets. Otherwise, the large number of $backpacks$ would result in huge memory usage. The maintenance includes the creation and deletion of the $backpacks$. The *source* is responsible for the creation of the $backpack$ for each packet, and the *sink* deletes the $backpacks$ of the packets that reach it. However, for the packets that are lost in the way, the *sink* cannot be aware of them. Thus we design a loss detection module to perform the detection of the lost packets and can delete the corresponding $backpacks$. Since the common implementation of VNF includes the container technology which shares the OS kernel with the server and the virtual machine (VM) technology which does not share the OS kernel, the loss detection module has two ways to detect the lost packets. The first way is suitable for the container VNFs. Leveraging the feature that the kernel function *kfree()* is called when a packet is lost in a container, we attach an eBPF program on the hook of the *kfree()* function. Every time the *kfree()* is called, we can know which packet is lost. For the VM VNFs, the kernel is not shared, so the packet loss in the VM does not trigger the *kfree()* in the server kernel. We preset a timeout value and then poll $backpacks$ periodically, if the time since the last update of the $backpack$ exceeds the timeout value, we determine that the packet corresponding to the $backpack$ has been dropped.

D. Supported Metrics

Here we give an introduction of the metrics supported by vNetRadar, these metrics are on a per-hop and per-packet level.

Throughput. Throughput can be used to measure the performance of transport protocols. For each VNF, over a period of time ΔT, we get the size of each packet and add it up to the total data volume A, then the traffic throughput of the VNF equals $A/\Delta T$.

Latency. Latency is an important factor affecting QoS. The supported latency includes the per-hop latency and the end-to-end latency. Since the virtual Ethernet pair used to connect VNFs is just a pointer exchange with very low latency, we consider the per-hop latency as $T_i - T_{i-1}$, where T_i is the timestamp when the packet leaves the i_{th} VNF. Similarly, the end-to-end latency is $T_i - T_1$. Since the VNFs are all on the same server, the time between the VNFs is synchronized.

Packet Loss Location. Locating where packet loss occurs plays a significant role in locating where the black holes and the congestion occur. Since the lost packet's metadata carries the device ID of the last node it passes through, we can know where the packet is dropped.

Path. Per packet path can help evaluate the traffic scheduling algorithms. There is an array P in the packet's metadata, for each VNF the packet passes through, we push the device ID of that VNF into P, thus enabling the path recording.

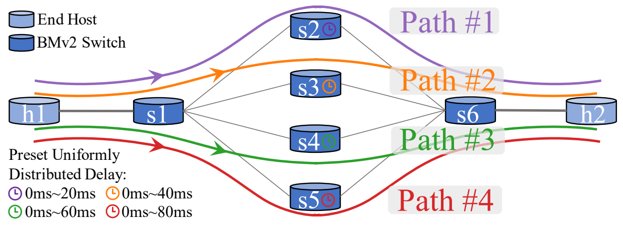

Fig. 3: The load balancing scenario, where s2, s3, s4, and s5 add a uniformly distributed delay to each packet that passes through it.

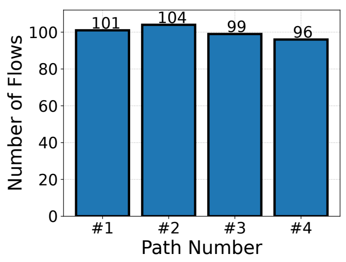

Fig. 4: vNetRadar measures the per-flow path exactly.

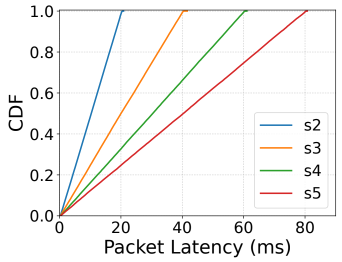

Fig. 5: vNetRadar accurately measures the distribution of the per-hop per-packet latency.

Other Metrics. Besides, more metrics such as flow completion time and latency jitter can be measured by carrying the desired metadata in the *backpacks* as required.

IV. Evaluation

We conduct experiments on a 64-core server with two Intel(R) Xeon(R) Gold 5218@2.30GHz CPUs and 128GB RAM. The OS is Ubuntu 20.04 with Linux kernel 5.13. We first use a load balancing experiment to demonstrate vNetRadar's application and verify its function, then three other experiments are conducted to evaluate vNetRadar's performance.

A. Use Case: Load Balancing in Virtual Network

Researchers often evaluate the load balancing algorithms through virtual network emulation, therefore, it will be helpful if metrics such as path and per-hop delay can be measured. We measure these two metrics of the traffic in a load balancing scenario[4] from the network emulation system, Kathará [2], to demonstrate vNetRadar's ability to get the fine-grained network-wide metrics. The topology of the scenario is shown as Fig. 3, where the BMv2 switch stands for Behavioral Model version 2, the P4 software switch[5]. We let h1 send 400 TCP flows with randomly generated pairs of source and destination TCP ports to h2, each flow has 500 packets, and s1 routes the flows to one of the four paths based on the hash value of the 5-tuple (i.e. the source TCP port, the destination TCP port and the 3-tuple) of the packet header. In addition, we use TC on s2, s3, s4, and s5 to let these VNFs have a delay with a uniform distribution ranging from 0ms to 20ms, 0ms to 40ms, 0ms to 60ms, 0ms to 80ms, respectively.

As shown in Fig. 4, vNetRadar reports that the number of flows on each of the 4 paths is close to 100, which proves that s1 completes the load balancing task correctly. Fig. 5 is plotted from the per-hop per-packet latency and shows the distribution of the latency for each packet as it passes through s2, s3, s4, and s5. We can see that most of the packets' latency follows a uniform latency distribution as expected, while a few packets have a slightly higher latency than the preset maximum latency due to queuing etc. These two results reported by vNetRadar are in line with expectations, proving that vNetRadar can correctly obtain fine-grained network-wide metrics.

[4]"the load balancing scenario," https://github.com/KatharaFramework/Kathara-Labs/tree/master/P4/05-ECMP

[5]"Behavioral model," https://github.com/p4lang/behavioral-model

B. Performance Evaluation

We compare vNetRadar with two typical schemes:

- *tcpdump* [12]. tcpdump is a common traffic mirroring scheme used in virtual networks. We carry a unique ID in the packet payload to help tcpdump perform the packet identification, thus can get network-wide metrics.
- *PPTMon* [11]. Similar to vNetRadar, PPTMon is also an eBPF-based scheme. Meanwhile, as stated in Section II-B, the mechanism of PPTMon is similar to the well-known INT. Thus PPTMon is representative to be compared. In our experiments, we use PPTMon's continuous mode (i.e. measure every packet) and disable its event filtering mechanism to get the fine-grained data.

As shown in Fig. 6, we use a testbed to evaluate the bandwidth, latency, and CPU overhead of the measurement schemes. The testbed is built on containers and consists of 2 end host nodes and N routers (containers configured with IP forward). We will specify the value of N in each experiment.

1) Bandwidth Overhead

Some schemes use extra bits to carry the metadata and bring the bandwidth overhead. To evaluate this overhead, we measure the goodput (i.e. the end-to-end effective transfer rate) on different measurement path lengths N, which is set from 1 to 10. The link bandwidths between the forwarding nodes are limited to 100 Mbps by TC, and a UDP traffic is sent from h1 to h2 at a speed greater than 100 Mbps. The schemes measure this UDP traffic to get its network-wide metrics. Since the performance of PPTMon is affected by the packet size, we evaluate the cases of packet sizes of 64 bytes and 1514 bytes.

Fig. 7 shows that vNetRadar and tcpdump have a low bandwidth overhead and enable high goodput when performing measurement tasks. As the path length increases, the UDP goodput of vNetRadar and tcpdump maintains the highest value, while the goodput of PPTMon gradually decreases. When the path length is 10 and the packet size is 64 bytes, the goodput of PPTMon decreases by 42.4%. This is due to PPTMon inserting a lot of extra bits into the packet, which takes up a large amount of bandwidth. Note that this bandwidth overhead is not only present in PPTMon, but also in other in-band telemetry schemes that insert extra bits into the packet.

In addition to this, we manually and carefully set the value of the MTUs when evaluating PPTMon, while no additional settings are made for the MTUs when evaluating vNetRadar, demonstrating that vNetRadar is easy-to-deploy.

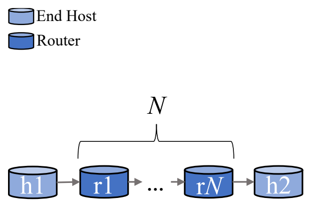

Fig. 6: The topology of the testbed consists of 2 end hosts and N routers.

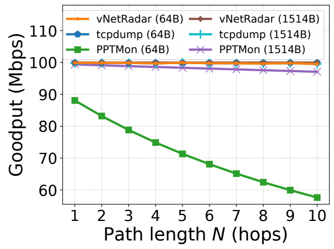

Fig. 7: vNetRadar has a low bandwidth overhead, the 64B and 1514B represent the packet size.

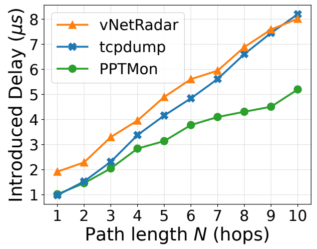

Fig. 8: vNetRadar introduces about 1 μs delay per hop, which is negligible.

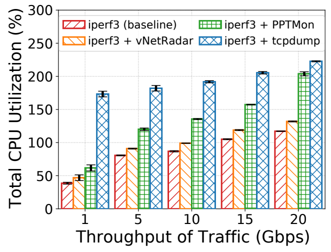

Fig. 9: vNetRadar is lightweight in terms of CPU overhead.

2) Introduced Delay

vNetRadar processes packets on the data path and thus introduces additional network delay. To evaluate the delay introduced by the schemes on different path lengths N, we set the path length N from 1 to 10. Then we run *ping -f* on h1 to flood 10^4 packets to h2 and get the RTT between h1 and h2. Since we only use the schemes to process packets in one direction, by subtracting the average RTT with/without the measurement scheme, we obtain the additional one-way delay introduced. As the resolution of the RTTs get from *ping -f* is just $1\mu s$, we repeat the above process 100 times to improve the accuracy of the evaluation result.

As shown in Fig. 8, vNetRadar achieves a similar delay performance to tcpdump which only performs copy operation on the data path, and PPTMon has a better delay performance since it has less BPF map I/O operations than vNetRadar. This is negligible and acceptable because vNetRadar introduces an average delay of only about $1\mu s$ per hop, which is an extremely small value compared to the ms-level latency of application traffic.

3) CPU Overhead

In this experiment, we evaluate the CPU overhead of the schemes when measuring traffic with different throughputs. We set N to be 1, then use *iperf3*[6] to send TCP traffic with different throughputs from h1 to h2. Since vNetRadar and PPTMon run in the kernel and are event-driven, it is difficult to measure their CPU overhead, we run *iperf3* with/without the measurement scheme to show the overhead. We get the total CPU utilization in a minute by the *psutil*[7] tool of Python, which means the sum of the CPU utilization of all cores. We repeat the above process 10 times.

As shown in Fig. 9, the CPU overhead introduced by vNetRadar ranges from 8.3% to 14.7% for different throughputs, which is much lower than tcpdump and PPTMon. This is because vNetRadar processes packets directly on the kernel datapath, whereas tcpdump copies packets to the user space, which introduces significant CPU overhead. Besides, PPTMon's CPU overhead is mainly due to submitting the packets' latency information from the kernel space to the user space upon each packet arrives at the last measurement hop, we believe the CPU overhead can be decreased if PPTMon changes the per-packet submission operation to batch submission.

[6]"iperf3," https://iperf.fr/
[7]"psutil tool," https://github.com/giampaolo/psutil

V. Conclusion and Future Work

In this paper, we propose vNetRadar, a virtual network measurement system that simultaneously meets four requirements: fine-grained, lightweight, network-wide, and easy-to-deploy. The core ideas of vNetRadar are the identification of packets without growing their size and the storage of metadata in *backpacks* in memory. Our evaluation results demonstrate that vNetRadar can easily achieve a comprehensive view of virtual networks with low overhead. Currently, vNetRadar only supports the measurement of virtual networks on the same server, and we will add support for multi-server based virtual networks in the future.

References

[1] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010, pp. 1-6.
[2] M. Scazzariello, L. Ariemma, and T. Caiazzi, "Kathará: A lightweight network emulation system," in *NOMS*, 2020, pp. 1-2.
[3] Y. Wu, K. Zhang, and Y. Zhang, "Digital twin networks: a survey," *IEEE Internet of Things Journal*, vol. 8, no. 18, pp. 13789-13804, 2021.
[4] J. Touch, "Updated specification of the IPv4 ID field," *Internet Engineering Task Force (IETF)*, pp. 2070-1721, 2013.
[5] P. Emmerich, D. Raumer, S. Gallenmüller, *et al.*, "Throughput and latency of virtual switching with open vswitch: A quantitative analysis," *Journal of Network and Systems Management*, vol. 26, no. 2, pp. 314-338, 2018.
[6] Y. Cao, A. Jain, K. Sharma, *et al.*, "When to use and when not to use BBR: An empirical analysis and evaluation study," in *IMC*, 2019, pp. 130-136.
[7] T. Mizrahi, G. Navon, G. Fioccola, *et al.*, "AM-PM: Efficient network telemetry using alternate marking," *IEEE Network*, vol. 33, no. 4, pp. 155-161, 2019.
[8] C. Kim, A. Sivaraman, N. Katta, *et al.*, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, 2015.
[9] R. Ben Basat, S. Ramanathan, Y. Li, *et al.*, "PINT: Probabilistic in-band network telemetry," in *ACM SIGCOMM*, 2020, pp. 662-680.
[10] Y. Zhao, K. Yang, Z. Liu, *et al.*, "LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets," in *NSDI*, 2021, pp. 991-1010.
[11] T. Van, J.-H. Yoo, and J. Hong, "PPTMon: Real-Time and Fine-Grained Packet Processing Time Monitoring in Virtual Network Functions," *IEEE Transactions on Network and Service Management*, vol.18, no.4, pp. 4324-4336, 2021.
[12] "tcpdump," https://www.tcpdump.org/.
[13] Y. Zhu, N. Kang, J. Cao, *et al.*, "Packet-level telemetry in large datacenter networks," in *ACM SIGCOMM*, 2015, pp. 479-491.
[14] M.A.M Vieira, M.S. Castanho, R.D.G. Pacífico, *et al.*, "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications," *ACM Computing Surveys*, vol. 53, no. 1, pp.1-36, 2020.