Measuring End-to-end Packet Processing Time in Service Function Chaining

Nguyen Van Tu, Jae-Hyoung Yoo, and James Won-Ki Hong

Dept. of Computer Science and Engineering, POSTECH, Pohang, Korea
Email: {tunguyen, jhyoo78, jwkhong}@postech.ac.kr

***Abstract*—Network Function Virtualization (NFV) is the key to enable rapid development and deployment of network services as well as simplicity and flexibility in network operations and management. To achieve the maximum benefit of NFV, monitoring the performance characteristics of Virtual Network Functions (VNFs) is crucial. Packet processing time is one of the most important performance metrics when it comes to VNF monitoring. In this paper, we present Packet Processing Time Monitoring (PPTMon) - a real-time, end-to-end solution for VNF packet processing time monitoring. PPTMon can provide per-hop monitoring for a single VNF as well as end-to-end monitoring for multiple VNFs in service function chains. PPTMon works by embedding timestamp information directly into the packets. PPTMon is implemented on top of extended Berkeley Packet Filter (eBPF) - a new Linux framework that allows high-speed packet processing. Our experiment results showed that PPTMon can monitor VNF packet processing time with high accuracy and negligible performance impact.**

***Keywords*—Real-time Network Monitoring, NFV, SFC, VNF Monitoring, eBPF, Measuring Packet Processing Time**

I. Introduction

For a long time, telco's network services have been deployed using physical proprietary middleboxes from vendors. These middleboxes implement various network functions such as switches, routers, firewalls, network address translations (NATs), intrusion detection/prevention systems (IDSs/IPSs), traffic classifiers, web accelerators, and load balancers. Because they are vendor-specific and proprietary, these middleboxes are very costly to deploy, operate, maintain, and upgrade. The process of deployment and upgrading is also slow and complex. In contrast, the rapid change in the network services requirements nowadays leads to a short life-cycle of physical middleboxes, thus the middleboxes need to be upgraded or replaced more frequently. 5G networks require a high degree of flexibility and agility of the network services [1], which traditional hardware middleboxes may not be able to provide. For many years, telcos have been paying high capital expenditure (CAPEX) and high operating expense (OPEX) using proprietary middleboxes, while not being able to provide flexibility or fast time to market.

Network Function Virtualization (NFV) [2] is an effort to tackle these problems. NFV decouples the hardware and software parts of the middleboxes, turns the network functions into plain software that can run on any industry-standard, commodity servers, thus called Virtual Network Functions (VNFs) [2]. NFV has a set of standards so that VNFs are vendor-neutral and can be easily chained together to provide useful services. Telcos have been gaining many benefits from applying NFV: lower deployment and maintenance cost, faster upgrade process, and more flexible in operations and management. Additionally, NFV also has positive effects on the vendor side. By defining standard hardware and focus on the software part, NFV also allows faster and easier VNF development and deployment, which reduces the time-to-market. New vendors have a chance to join the market, old vendors also need to join the NFV development if they do not want to fall behind and lose their customers.

Monitoring is crucial to the deployment, operations, and management of any systems including NFV. In the case of an NFV system, one of the most important key performance indicators is the packet processing time of VNFs, i.e., from the time when a packet goes into a VNF to the time when the packet exits the VNF. Monitoring packet processing time helps to ensure that VNFs are working correctly with the expected performance, and helps to debug when there are anomalies in the network services. Packet processing time is an important metric in service level agreements. The monitoring data can also be used as inputs for VNF modeling or machine learning-based NFV management systems. However, despite the importance of the packet processing time metric, monitoring VNF packet processing time is still a challenge.

In this paper, we present **P**acket **P**rocessing **T**ime **Mon**itoring (PPTMon) - a real-time, light-weight, and end-to-end method for packet processing time monitoring. At the VNF level, when a packet passes a VNF, the packet processing time value is calculated as the difference between ingress timestamp and egress timestamp. At the service function chain (SFC) level, PPTMon attaches PPTMon header and processing time data to a packet at the first VNF in the chain, adds new processing time data of the subsequent VNFs to the packet when the packet passes each VNF, and finally extracts PPTMon header and all PPTMon data at the final VNF in the chain. Because adding a custom header to a packet can potentially affect how the VNF processes the packet, the PPTMon header format is carefully considered so that PPTMon is transparent to the legacy PPTMon-unaware VNFs. PPTMon is implemented using extended Berkeley

978-3-903176-31-7

Packer Filter (eBPF) [3] - a new Linux kernel framework that allows high-performance packet processing. We evaluated PPTMon in real NFV environment using OpenStack [4]. The evaluation results showed that PPTMon has an average of 3.6% performance reduction when doing stress tests with various VNFs and negligible performance reduction when doing real use-case tests with SFCs.

This work extended our recent work [5] in several aspects. Firstly, our previous work considers packet processing time monitoring in a single VNF (per-hop monitoring). This work extends [5] to the context of service function chaining. In particular, this work supports both per-hop monitoring of single VNF and end-to-end monitoring of multiple VNFs in an SFC. The end-to-end approach has differences in design, implementation, and advantages compared to applying [5] repeatedly to multiple VNFs, which will be explained in detail throughout this paper. Secondly, [5] only supports TCP, while this work supports both TCP and UDP. Thirdly, we conducted extensive experiments to verify how PPTMon works in real-world SFCs.

The remainder of the paper is organized as follows. In Section II, we present the background and related work. In Section III, we present the detailed algorithm, header format, and implementation of PPTMon. Section IV presents the evaluation results of PPTMon and discusses PPTMon's limitations and future improvements. Finally, Section V concludes this paper.

II. Related Work

In this section, we present related work about monitoring packet processing time and VNF performance in general. We also cover eBPF and discuss its performance advantage.

A. Packet processing time monitoring

There are several methods and research efforts to monitor packet processing time in VNFs. A very naive method is to capture ingress and egress packets with timestamps using tools such as tcpdump[1], then inspect the captured packets to get the processing time. However, capturing every packet inside a VNF incurs very high overhead, thus greatly reduces the throughput and increases the latency of VNF. Hence, this method is impractical in production systems and only suitable for debugging and offline testing purposes.

NFVPerf [6] moves the packet capture functions out of the VNF by mirroring all VM-to-VM traffic to a central processing node. NFVPerf uses deep packet inspection to analyze the traffic metric such as delay and CPU usage. Using this approach, NFVPerf can minimize the negative effect on the VNF, at the cost of extra network bandwidth and an extra dedicated node for NFVPerf processing.

KOMon [7] eliminates the packet capture overhead by using a different approach. KOMon uses a kernel module to inspect traffic at the ingress point of a VNF, saves the payload hash and timestamp into a queue. At the VNF egress, KOMon tries to match the payload hash value of the packet with the ingress payload hash. If the values are the same, then KOMon subtracts the timestamp to get the packet processing time. KOMon thus can provide real-time data with low overhead. However, KOMon has some limitations. KOMon only works with VNFs that process packets in the FIFO model and does not change the packet payload, KOMon is implemented as a custom kernel module, thus it has potential security and stability issues. Also, the payload hash method can potentially cause incorrect measurements in the case of packets with the same payloads (e.g., re-transmission, packet dropped).

SymPerf [8] uses code analysis to predict the performance of a VNF during run-time. Therefore, SymPerf does not have any performance impact on the VNFs at all. However, SymPerf requires access to the VNF source code, which is not always available, e.g., black-box VNFs, or VNFs from third-party vendors. Also, the unpredictable events, such as anomalies that increase packet processing time, can not be captured during the run-time.

In [9], authors use ICMP echo request and reply packets to measure the packet processing time of a physical host. The method can monitor delay without any software injection on the host. However, because the method requires a special packet-capture card with timestamp function, it is only available for physical host, not VM. Also, the processing time of ping packets does not necessarily represent the processing time of other packets.

B. Network monitoring

Network monitoring is a much broader topic than VNF packet processing time monitoring, therefore we only cover two cases that inspired our work here: TPP [10] and INT [11, 12]. TPP and INT are designed for fine-grain and end-to-end monitoring of hardware switches. They work by attaching custom monitoring data directly to the packets. The custom data are the switch state reported at the time the packet passed by, such as hop latency, queue size and occupancy, or link utilization. As a packet passes switches in its path, the custom monitoring data are added up to the packet. At the final switch, all custom monitoring data is extracted and removed from the packet. The extracted data can be processed locally or sent to a central collector for further processing.

The advantages of TPP and INT are its fine-grain and end-to-end monitoring: they provide the exact path of the packet, the view of the whole network from the viewpoint of that particular packet when it travels its path, at the packet-level granularity. This not only provides detail and complete view of the network but also is useful for debugging and troubleshooting the network. However, because TPP and INT monitor networks at the packet-level, they have very-high overhead on switch workload and network bandwidth. Therefore, TPP and INT are only suitable for hardware switches with on-switch programmable ICs, such as FPGA. The implementation can

[1] tcpdump, https://www.tcpdump.org

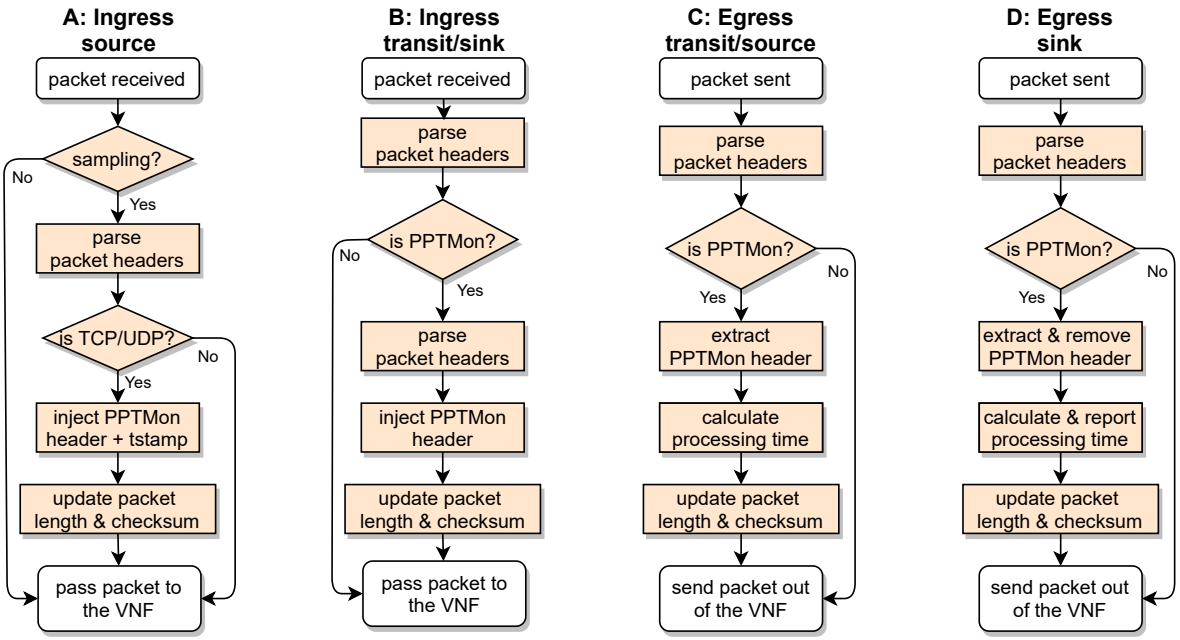


Fig. 1: PPTMon algorithm

be done using a low-level language supported by the IC, such as Verilog, or a target-independent language, such as P4 [13].

In this work, we applied a similar approach with TPP and INT to PPTMon: the packet processing time is attached to a packet at the first VNF, added up when the packet passed subsequent VNFs, and extracted at the final VNF in the chain. However, the similarity is only in the abstract idea, the algorithms and implementations are different because of different targets: TPP and INT target hardware switches, while PPTMon targets virtual network functions. Also, because PPTMon is running as software on commodity servers instead of programmable hardware switches, we applied sampling to reduce the performance penalty, with the trade-off of lower granularity.

Compared to our recent work in [5], This work provides an end-to-end view instead of an independent per-VNF view. While [5] only provides independent per-VNF packet processing time, this work also provides the exact path of the monitored packet with a complete view of packet processing time of VNFs in the packet's path. This provides a better view of the network and SFCs. End-to-end monitoring is also especially helpful in debugging/troubleshooting SFCs.

C. extended Berkeley Packet Filter

extended Berkeley Packet Filter (eBPF) [3] is a Linux kernel framework that allows attaching user-supplied programs to a kernel event type. An eBPF program lives inside the Linux kernel as a light-weight virtual machine and is called when the event happens. An eBPF program is written in a restricted subset of C language and is compiled to eBPF instruction set which is mapped closely to the hardware instruction set. Also, eBPF compiler supports just-in-time compilation. Therefore, eBPF provides performance closes to the native C code. Although running inside the kernel, unlike a custom kernel module, eBPF programs are verified using kernel eBPF verifier during the compile-time to ensure the safety and security of the kernel. Also, the exposed interface to write eBPF program is stable, thus programmers do not need to worry about maintaining the compatibility with the new kernel versions like a custom kernel module.

eBPF can be used for high-performance packet processing [14]. In networking, an eBPF program can be attached to several layers in the kernel networking stack, such as traffic control or socket layer, and is called when a packet event happens, such as a packet received or sent. Because eBPF is well integrated with the kernel networking stack, the post-processed packet can be passed to the kernel stack for processing as normal. Compared to the user-space equivalent program, an eBPF program does not have the overhead of kernel-user space context switching. Also, the kernel creates a copy of the packet when sending it to the user-space program, while the eBPF program processes the packet in-stack and does not require packet copy.

III. Design and Implementation

In this section, we show the detailed algorithm and workflow of PPTMon. We present the PPTMon header format and discuss why we chose the design. Then we present the implementation detail of PPTMon with eBPF.

A. PPTMon's algorithm

Fig. 1 shows the detailed algorithm of PPTMon. For easy understanding, we explain PPTMon's algorithm in two levels: VNF level, and SFC level.

At the VNF level, we consider how PPTMon calculates packet processing time inside a VNF. At the early stage when a packet arrives at a VNF, PPTMon inspects the packet and checks whether the condition to inject the PPTMon data is met. PPTMon then adds the VNF ID and the current local timestamp of the VNF to the packet, updates the equivalent packet length and checksum, then passes the packet to the kernel networking stack. After the packet is processed by the VNF function, the packet is inspected by PPTMon before sent out. If the packet has a PPTMon header, the processing time is calculated as the difference between the current time and the stored timestamp, then the PPTMon timestamp field in the packet is replaced by the calculated packet processing time.

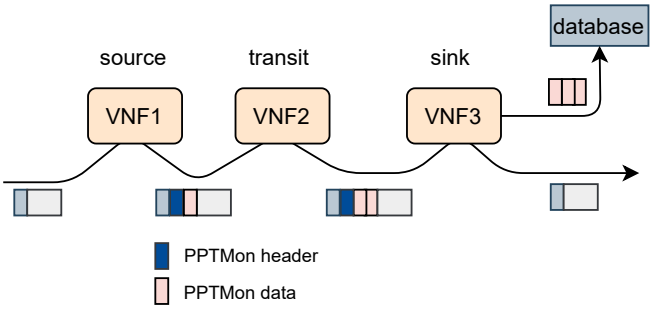


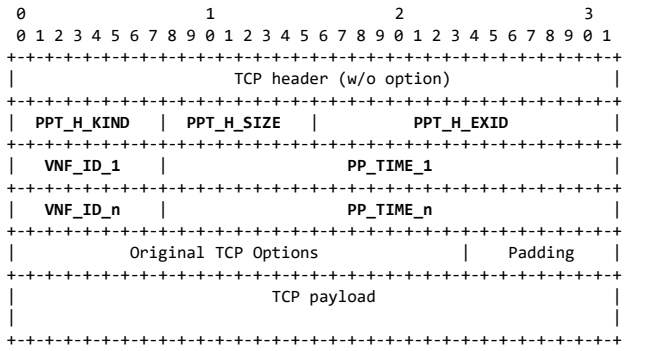
Fig. 2: PPTMon in service function chain

At the SFC level, we consider how PPTMon works at different nodes in the SFC. The abstract flow of a packet with PPTMon activated is shown in Fig. 2. In each VNF, PPTMon can work in one of four modes: source, transit, sink, and source-sink. In the source mode, which is activated if the VNF is the first one in the service chain, PPTMon does the sampling to selectively activate PPTMon monitoring for packets, e.g., one activation per second. If a packet is selected, the packet header is parsed, then PPTMon header and packet processing time data with the VNF ID are inserted into the packet. In the transit mode, which is activated if the VNF is in the middle of the chain, PPTMon parses the packet and detects whether the packet has a PPTMon header. If the PPTMon header is presented, the ID of current VNF with its packet processing time is added to the PPTMon data. In the sink mode, which is activated if the VNF is the final one in the chain, PPTMon extracts PPTMon header and all PPTMon data and restores the original packet before the packet is forwarded to the destination. The extracted monitoring data can be stored in a database for further queries. If there are only two VNFs in the SFC, there is no transit node. If there is only one VNF in the SFC, PPTMon runs in source-sink mode, where it does the function of both the source node and sink node.

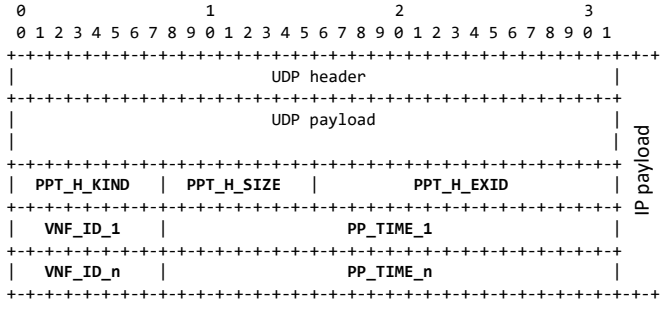
Running PPTMon for every packet can cause a high overhead on CPU usage, reduce the throughput, and produce a huge amount of monitoring data, especially for high-throughput VNFs with hundreds of thousands of packets per second. Therefore, we use sampling for PPTMon and leave the user to choose the sampling rate. When a new packet arrives, PPTMon simply checks the time passed from the last sampling, and if it exceeds the sampling period, a new PPTMon header is inserted into the packet.

B. PPTMon's header format

PPTMon header and data format need to be carefully designed because it is exposed to VNF and can potentially change the way VNF processes the packet. In PPTMon, the header and data are added to the packet as a TCP or UDP option field following the IETF spec [15, 16] so that PPTMon is transparent to the VNFs. PPT_H_KIND is set to 254, which is defined as an experimental option and should be ignored by the PPTMon-unaware VNFs, thus making PPTMon transparent to these VNFs. PPT_H_SIZE is the total length of the PPTMon header in bytes. VNF_ID is the one-byte ID of the current running VNF. One byte can address 256 IDs, which should be enough for any SFC. PP_TIME is the measured packet process time, which is a 24 bits value with microsecond unit. The maximum packet processing time can be stored is $2^{24} \approx 16$ seconds, which we consider more than enough for a normal behaving VNF. If the packet processing time is more than several seconds, the VNF might behave abnormally or have errors, thus it needs to be investigated.



(a) PPTMon as TCP option



(b) PPTMon as UDP option

Fig. 3: PPTMon header as TCP/UDP option

The format of PPTMon as TCP option is shown in Fig. 3-a, following the standard TCP option format [15]. The format of PPTMon as UDP option is shown in Fig. 3-b, following the IETF draft of UDP option [16]. In a UDP packet, both the IP length field and UDP length field contains the header and data length. The redundant information can be used to indicate extra UDP option at the end of the UDP packet payload and it is

permissible [16]. Note that while it is possible to put PPTMon as an IP option, several VNFs, such as iptables NAT, modify the IP header and may drop the IP option, thus losing the PPTMon header and data during the process.

Although applying a new option kind usually requires proper registration [17], we designed PPTMon so that it can be used in production without registering a new option kind. Firstly, PPTMon header only exists when the packet is inside the VNF chain, particularly from the first VNF in the chain to the last VNF in the chain, thus PPTMon does not expose to the rest of the network and the Internet. Secondly, we follow the rule to share the usage of TCP experimental option [18] in a controller environment by setting a dedicated PPT_H_EXID for PPTMon. [18] is also applied to UDP option.

C. Implementation

PPTMon is implemented in eBPF using BCC[2] - a framework for compiling and running eBPF programs. The eBPF code is attached to the Traffic Control (TC) *clsact* [19] at both ingress and egress, as shown in Fig. 4. Note that the VNF can also live in kernel space, such as iptables firewall[3]. The user space module of PPTMon handles the processing of attaching and detaching eBPF programs to the kernel. It also handles the packet processing time reported by the PPTMon eBPF egress. The implementation of kernel-user space parts of PPTMon following the fast-slow path architecture from our previous work [20].

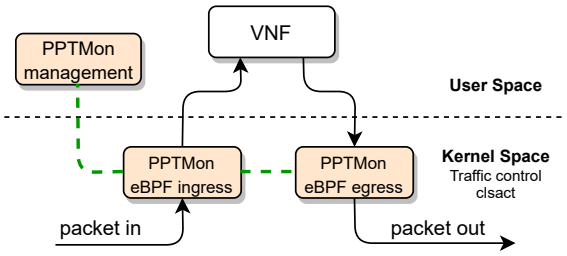


Fig. 4: PPTMon with traffic control clsact

In a precise definition, PPTMon measures the time from the TC *clsact* ingress, when Linux packet buffer *skb* is just created in the kernel; to the TC *clsact* egress, when the packet is already sent back to the kernel and is about to be sent out of the VNF. This measured time is the sum of the plain VNF processing time and the time Linux kernel handles the packet. The time Linux kernel handles the packet is negligible compared to the VNF processing time in normal cases, but it can be significant at high-speed VNF (e.g., iptables firewall), or when there are anomalies. Hence, PPTMon measured packet processing time actually provides a better metric to evaluate the state of the VNF than the plain VNF processing time.

Because the packet size grows as PPTMon header and data are added when the packet travels the service chain, the Maximum Transmission Unit (MTU) size of the NIC needs to be planned in advance so that there is enough room for PPTMon header and data. Also, because the TCP option size is limited, PPTMon needs to check in advance whether there is enough room in the option field before actually adding PPTMon data.

In this work, PPTMon is implemented using eBPF. However, the same approach and algorithm can be applied for other network stacks, such as DPDK[4]. Besides, in the case of using old Linux kernels that do not support eBPF, PPTMon can be implemented as a kernel module like KOMon [7]. PPTMon's source code is available on GitHub[5].

IV. Evaluation

In this section, we present the evaluation of three aspects of PPTMon: the average reported processing time, the effect of sampling rate to the performance, and the performance when using PPTMon with the popular VNFs, such as firewall, NAT, IDS, DPI, and load balancer. Then, we present how PPTMon performs with real-world SFCs: the first chain is firewall → load balancer → web server, and the second chain is firewall → IPS → load balancer → web server. We also discuss the limitations of PPTMon and possible improvements and future work.

A. Testbed setup

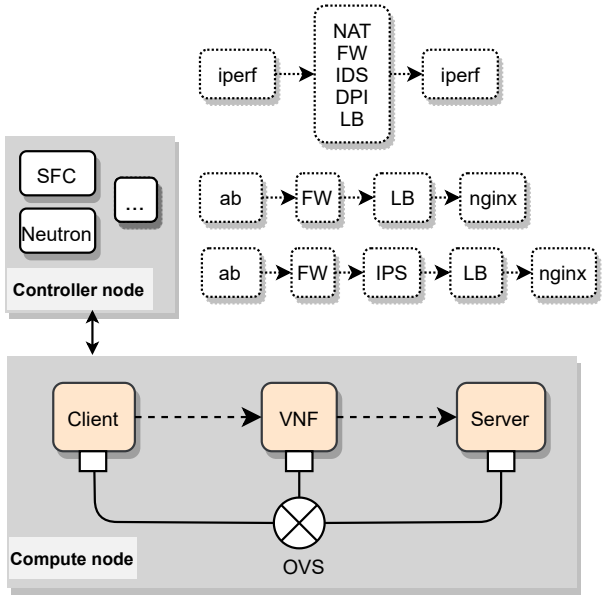


Fig. 5: PPTMon testbed with OpenStack

To evaluate PPTMon and observe how PPTMon performs in real use cases, we deployed our test scenarios using OpenStack [4], an open standard platform for cloud computing and NFV. The testbed deployment is shown in Fig. 5. We used a separate controller node to host OpenStack controller services so that

[2]BPF Compiler Collection, https://github.com/iovisor/bcc
[3]iptables, https://www.netfilter.org/projects/iptables/index.html
[4]DPDK, https://www.dpdk.org
[5]PPTMon, https://github.com/dpnm-ni/ppt-mon

their CPU usage does not affect the evaluation result. All VMs and VNFs were deployed on one OpenStack compute node. Although there are usually multiple compute nodes in a real operation environment, using only one compute node in this testbed does not affect the purpose of our experiments. We used Open vSwitch (OvS)[6] for inter-VM and VNF networking. All VMs and VNFs ran Ubuntu server 19.10 with kernel v5.3. Each VNF was allocated 2 vCPU and 2 GB of RAM. The OpenStack compute node is a Dell R610 server with 2 Intel Xeon X5650 CPUs and 24GB RAM, distributed in 2 NUMA nodes, with hyper-threading enabled.

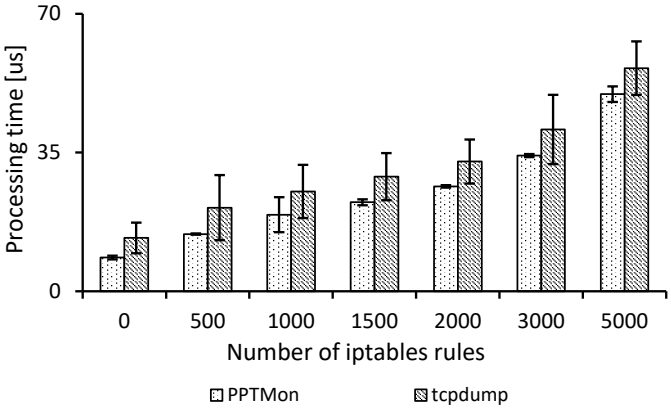
In all the tests, three VMs are used: two VMs ran as client and server, one VM was used for VNF deployment. In the case of transparent VNFs (firewall, IDS, DPI), OpenStack networking-sfc[7] was used to create the network function chains to forward packets via the VNF.

B. Average processing time

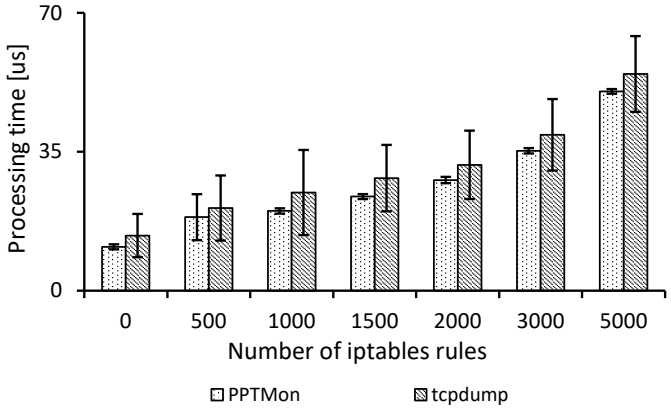
In this scenario, we compared the values measured by PPTMon and tcpdump. Although PPTMon accuracy is ensured by the algorithm, the value only represents the delay time of the sampled packets, not all packets. Thus, we used tcpdump to capture packets with their timestamps to see whether PPTMon could represent the average packet processing time of the VNF. To the best of our knowledge, there is no available method to do sampling in tcpdump (or packet capture in general) that captures the same packet in both ingress and egress, which is necessary to calculate packet processing time. Thus, we captured all packets in the case of tcpdump.

In the test, both client and server ran iperf3[8] and exchanged TCP or UDP traffic. The VNF ran both PPTMon and tcpdump. When we did not limit iperf3's bit-rate, the maximum throughput when using tcpdump and PPTMon was 0.6 Gbps and 1.8 Gbps, respectively. Running without bit-rate limitation also produced too much data in the case of tcpdump. Therefore, we ran iperf3 at a low bit-rate (500 Kbps) so that the overhead of tcpdump would not affect the accuracy of the measurement. The VNF is iptables firewall with an increasing number of rules. We used dummy rules that did not match the iperf3 traffic, i.e., all packets need to be matched again all rules before going to the default rule, which forwards packets to the server. For each VNF setup, we ran the iperf3 traffic in 20 seconds. The PPTMon sampling period was set to 1 second.

The results are shown in Fig. 6. Intuitively, the processing time measured by both PPTMon and tcpdump increased when the number of flow rules increased. In all the cases, the average packet processing time reported by PPTMon was lower than the one from tcpdump. The average of the differences was 6.5 μs with a standard deviation of 0.9 μs for TCP traffic, and 3.8 μs with a standard deviation of 0.9 μs for UDP traffic, which was quite stable. While both PPTMon and tcpdump work at the low level of the kernel stack, the differences



(a) TCP traffic



(b) UDP traffic

Fig. 6: Average packet processing time reported by PPTMon and tcpdump for TCP and UDP traffic

are caused by the overhead of tcpdump. Because the value reported by tcpdump is the average of all packets, we conclude that PPTMon reported measurements can represent the average packet processing time.

This experiment also showed the baseline minimum latency of the kernel. In the case there is no firewall rule, the VNF just forwards the packet to the server right inside the kernel. The average delay measured by PPTMon was 8.5 μs with the standard deviation of 0.5 μs for TCP traffic, and 11 μs with the standard deviation of 0.6 μs for UDP traffic. The result was similar when the tcpdump is disabled.

Looking at the standard deviation of tcpdump for both TCP and UDP traffic, the value was higher than the equivalent of PPTMon in most cases, except the cases of 1000 iptables rules for TCP and 500 iptables rules for UDP. There are several reasons for this result. One reason is that PPTMon adds delay to the packet when the packet is sampled for measurement, and tcpdump also captures that delay. However, this delay does not affect PPTMon accuracy, since the experiment showed that the PPTMon reported measurement can represent the average packet processing time.

The other reason that caused high standard deviation of

[6]OvS, https://www.openvswitch.org

[7]networking-sfc, https://opendev.org/openstack/networking-sfc

[8]iperf3, https://iperf.fr

tcpdump reported values is that, naturally, there are uncertainties in the network and some packets had much higher processing time than the average. In some cases such as the cases of 1000 iptables rules for TCP and 500 iptables rules for UDP, these packets were sampled by PPTMon, which caused a high standard deviation in PPTMon reported values. In other cases, PPTMon did not sample these packets. This showed the drawback of sampling: it reduces the granularity of the measurement.

C. Effect of sampling rate on throughput

In this test, we measured how the sampling rate of PPTMon affects performance. The VNF ran iptables firewall with no rules. The client and server ran iperf3. We recorded the average TCP throughput reported by iperf3 when the sampling rate is changed. Each run lasted 20 seconds and was repeated 5 times. The results are shown in Fig. 7.

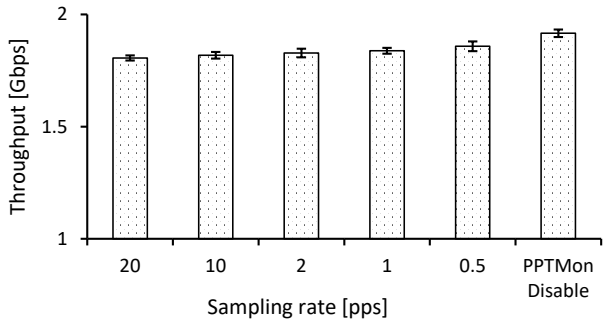


Fig. 7: Average throughput when changing the PPTMon sampling rate

The average throughput was slightly reduced when the sampling rate increased, with 5.7% reduction at 20 samples per second and 3% at 0.5 samples per second. We consider 5.7% is a small number, especially when the VNF is very lightweight: it just forwards the packets to the server directly in the kernel space. With more resource-intensive VNFs such as IPS, the throughput reduction caused by PPTMon is negligible, as will be shown in Section IV-D.

This experiment suggests two ways of using PPTMon. If users want to get the exact value of the sample packets every N sec (e.g., 1 sec), then the sampling rate can be set to $1/N$ packet per second (pps) (e.g., 1 pps). If a user wants to get the average processing time of last N sec, then the user can set the packet sampling rate at $10/N$ pps (e.g., 10 pps for $N = 1$ sec), then report the average of the last 10 measurement values. The throughput difference between the sampling rate of 1 pps and 10 pps is only 1%. Thus, there is no real performance disadvantage.

D. Throughput test with various VNFs

In this test, we measured the maximum throughput of various VNFs when PPTMon is enabled and disabled. The VNF collection includes iptables firewall, Suricata[9] IPS, nDPI[10] DPI, iptables NAT, and IPVS[11] load balancer. The detailed configuration of each VNF is shown in Table I. When enabled, PPTMon's sampling period is set to 1 sec.

TABLE I: VNF configurations

VNF type	Software	Configuration
firewall	iptables netfilter	20 non-matching rules
DPI	nDPI	nDPI reader v3.2 stable with iperf3 protocol detection
IPS	Suricata	Suricata v4.1 in IPS mode with default rule set
NAT	iptables netfilter	full NAT mode
load balancer	IPVS	IPVS load balancer in destination NAT mode

In the case of firewall, IPS, DPI, and NAT, both client and server ran iperf3 and we recorded the average throughput. Each run lasted 20 seconds and was repeated 5 times. The result is shown in Fig. 8, except for the load balancer test.

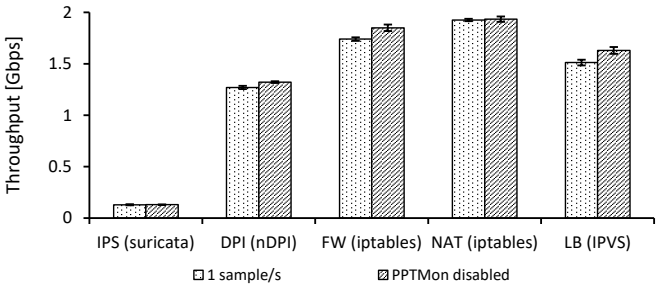


Fig. 8: Throughput when enable/disable PPTMon with various VNFs

In general, the throughput was slightly reduced when PPTMon is enabled. The percentages of throughput reduction were 0.6%, 3.9%, 5.9%, 0.4% and 7.2% for IPS, DPI, FW, NAT and LB, respectively. The effect of PPTMon overhead on throughput was reduced when the VNF is more resource-intensive and was increased when the VNF is more lightweight, except the case of NAT. In the case of iptables NAT, which is a lightweight VNF, the throughput reduction of PPTMon is only 0.4%. The average throughput reduction of all VNFs was 3.6%, and we consider it a small overhead.

E. Real SFC use cases

In this test, we deployed SFCs that are used in real-world and measured the maximum throughput of these SFCs when PPTMon is enabled and disabled. Two SFCs were evaluated: SFC1 is iptables firewall → IPVS load balancer → Nginx web server, and SFC2 is firewall → Suricata IPS → IPVS load balancer → Nginx web server, as shown in Fig. 5. The client

[9]suricata, https://suricata-ids.org
[10]nDPI, https://github.com/ntop/nDPI
[11]IPVS, http://www.linuxvirtualserver.org/software/ipvs.html

ran Apache bench[12] - a HTTP benchmark tool, and the server ran nginx[13] as a web server. We then recorded the average number of HTTP requests per second. In each run, the client sent 10000 requests to the server and the run was repeated 5 times.

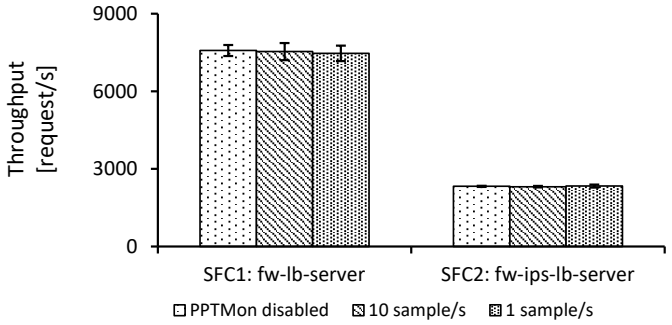


Fig. 9: Throughput when enable/disable PPTMon with real SFCs

The result is shown in Fig. 9. Suricata was the bottleneck that caused much lower throughput in the SFC2 compared to the SFC1. In both SFCs, the difference when enable and disable PPTMon was much less than the standard deviation of each measurement. PPTMon had a negligible impact on the throughput in both SFC tests.

F. Limitations and improvements

1) Limitations: Storing PPTMon's data in the TCP/UDP option field implies one limitation of PPTMon: it does not work for VNFs which drop the TCP/UDP option header. However, many of the VNFs do not modify the packets (e.g., firewall, DPI, IDS, and IPS), hence PPTMon works well with these VNFs. For the VNFs which modify the packets, PPTMon will work as long as the TCP/UDP option field is not removed. For example, PPTMon does not work with an encryption VNF if it encrypts IP layer, but will work if it encrypts layer 4 or higher. Another example is that some load balancers, such as HAProxy[14], modify the TCP layer and drop the PPTMon header; while other load balancers, such as IPVS, keep the TCP header. Thus, PPTMon does not work with HAProxy, but it works with IPVS.

In the case of UDP traffic, adding UDP option might cause issues in IPS/IDS systems that consider the miss-match between IP length field and UDP length field as a potential security compromise, e.g., Alcatel-Lucent's "Brick" Intrusion Detection System [16]. A workaround solution is to allow non-matching IP/UDP length in the IPS/IDS.

These are limitations users may face when using PPTMon in some special cases of legacy VNFs. This is hard to build a solution that works for all cases due to the heterogeneity of VNFs. If developers create new VNFs, then they just need to keep the PPTMon header and data in the packets to make PPTMon and the VNFs work together.

2) Improvements: We propose two directions to improve PPTMon. One direction is to modify PPTMon so that it can run in the physical host where VNF VMs are located instead of running directly inside the VMs. PPTMon can attach its eBPF ingress and egress programs to the virtual NIC on the host side. When running PPTMon in the host, all requirements (e.g., OS kernel) moved from the guest side to the host side. Therefore, running PPTMon in the host or guest complement each other, because they put the environmental constraints on different targets. PPTMon may also run less efficiently in a VM than in a physical host because of the virtualization overhead.

Another direction is to expand the measurements to other metrics such as CPU usage, NIC queue occupancy, and end-to-end delay. The only modification needed is a method to get and append these values to PPTMon, as the main mechanism to transport monitoring data should be kept the same. In the case of end-to-end delay, a time synchronization method should be used to ensure time synchronization between VNFs. This could provide a more complete solution for NFV monitoring and troubleshooting.

V. Conclusion

In an NFV system, monitoring VNFs, particularly the VNF packet processing time, is crucial for NFV operations and management. In this paper, we proposed PPTMon - a method for real-time, light-weight, and end-to-end packet processing time monitoring. At the VNF level, PPTMon works by attaching timestamp data to the packet header at the VNF ingress, then calculating the processing time at the VNF egress. At the SFC level, PPTMon adds the packet processing time monitoring data of the VNF to the packet when the packet passes each VNF, then extracts all monitoring data at the final VNF in the chain. We presented the design and implementation of PPTMon. The evaluation results showed that PPTMon is accurate and light-weight: PPTMon has an overhead of 3.6% VNF throughput reduction on average in stress tests with various VNFs and has negligible throughput impact on real SFC use cases.

We have several directions to improve PPTMon, as discussed in Section IV-F. Firstly, we plan to modify PPTMon to support monitoring from the side of the physical hosts. Secondly, we plan to develop PPTMon as a generic end-to-end monitoring platform to monitor not only packet processing time but also other metrics such as CPU usage or NIC queue utilization.

Acknowledgments

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (2018-0-00749, Development of Virtual Network Management Technology based on Artificial Intelligence).

[12]ab, https://httpd.apache.org/docs/2.4/programs/ab.html
[13]nginx, https://www.nginx.com
[14]HAProxy, http://www.haproxy.org

REFERENCES

[1] A. Gupta and R. K. Jha, "A Survey of 5G Network: Architecture and Emerging Technologies," *IEEE Access*, vol. 3, pp. 1206–1232, 2015.

[2] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, "Network Function Virtualization: State-of-the-Art and Research Challenges," in *IEEE Communications Surveys Tutorials*, vol. 18, 2016, pp. 236–262.

[3] A. Starovoitov, "BPF – in-kernel virtual machine," *Linux Kernel Developers' Netconf*, 2015.

[4] "Openstack." [Online]. Available: https://www.openstack.org

[5] N. Van Tu, J. Yoo, and J. W. Hong, "Real-time Monitoring of Packet Processing Time for Virtual Network Functions," in *2020 Asia-Pacific Network Operations and Management Symposium (APNOMS)*, Sept. 2020, *Accepted to appear*.

[6] P. Naik, D. K. Shaw, and M. Vutukuru, "NFVPerf: Online performance monitoring and bottleneck detection for NFV," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2016, pp. 154–160.

[7] S. Geissler, S. Lange, F. Wamser, T. Zinner, and T. Hoßfeld, "KOMon — Kernel-based Online Monitoring of VNF Packet Processing Times," in *2019 International Conference on Networked Systems (NetSys)*, 2019, pp. 1–8.

[8] F. Rath, J. Krude, J. Rüth, D. Schemmel, O. Hohlfeld, J. A. Bitsch, and K. Wehrle, "SymPerf: Predicting Network Function Performance," in *Proceedings of the SIGCOMM Posters and Demos*, ser. SIGCOMM Posters and Demos '17, 2017, p. 34–36.

[9] K. M. Salehin and R. Rojas-Cessa, "Measurement of packet processing time of an Internet host using asynchronous packet capture at the data-link layer," in *2013 IEEE International Conference on Communications (ICC)*, 2013, pp. 2550–2554.

[10] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, 2014, p. 3–14.

[11] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, 2015.

[12] The P4.org Applications Working Group, "In-band Network Telemetry (INT) Dataplane Specification," February 2020. [Online]. Available: https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_0.pdf

[13] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, 2014.

[14] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network service with eBPF: Experience and lessons learned," *High Performance Switching and Routing (HPSR)*, 2018.

[15] J. Postel, "Transmission Control Protocol," RFC Editor, STD, September 1981. [Online]. Available: https://tools.ietf.org/html/rfc793

[16] J. Touch, "Transport Options for UDP," RFC Editor, Tech. Rep., 2019. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tsvwg-udp-options

[17] S. Bradner and V. Paxson, "IANA Allocation Guidelines For Values In the Internet Protocol and Related Headers," RFC Editor, BCP, March 2000. [Online]. Available: https://tools.ietf.org/html/rfc2780

[18] J. Touch, "Shared Use of Experimental TCP Options," RFC Editor, RFC, August 2013. [Online]. Available: https://tools.ietf.org/html/rfc6994

[19] D. Borkmann, "Advanced programmability and recent updates with tc's cls bpf," *Proc. NetDev*, vol. 1, 2016.

[20] N. Van Tu, J. Yoo, and J. W. Hong, "Accelerating Virtual Network Functions with Fast-Slow Path Architecture using eXpress Data Path," *IEEE Transactions on Network and Service Management*, pp. 1–13, June 2020.