A Caching SFC Proxy Based on eBPF

Marco Haeberle*, Benjamin Steinert*†, Michael Weiss*, Michael Menth*
* University of Tuebingen, Chair of Communication Networks, Tuebingen, Germany
† University of Tuebingen, Zentrum für Datenverarbeitung, Tuebingen, Germany
Email: {marco.haeberle,benjamin.steinert,menth}@uni-tuebingen.de
michael-tobias.weiss@student.uni-tuebingen.de

***Abstract*—Service Functions (SFs) are intermediate processing nodes on the path of IP packets. With SF chaining (SFC), packets can be steered to multiple physical or virtual SFs in a specific order. SFC-unaware SFs can be used flexibly but they do not support SFC-specific encapsulation of packets. Therefore, an SFC proxy needs to remove the encapsulation of a packet before processing by an SFC-unaware SF, and to add it again afterwards. Such an SFC proxy typically runs on a server hosting virtual network functions (VNFs) that serve as SFs. Simple SFC proxies adapt a flow-specific static header stack. That is, each VNF requires an own SFC proxy, and the proxy cannot be extended to support per-packet metadata in the SFC encapsulation. The caching SFC proxy presented in this work caches packet-specific headers while packets are processed by a VNF, i.e., packet-specific header information is preserved. We present concept, use cases, and an eBPF-based implementation of the caching SFC proxy. In addition, we evaluate the performance of a prototype.**

I. Introduction

Service function chaining (SFC) steers defined traffic through a specific set of network functions (NFs) which are also called service functions (SFs) in that context. Traffic is classified at the network edge and the classification decision is encoded in additional SFC headers. This information is leveraged to forward the packets along the desired list of SFs. SFC is preferentially combined with software-defined networking (SDN) and network function virtualization (NFV) so that virtual NFs (VNFs) are used as SFs instead of physical appliances. This permits rapid changes to the sequence of SFs applied to a flow as both the path of the packets and the SFs can be managed by software-defined control.

There are many competing SFC solutions, e.g., the Network Service Header (NSH) [1], Segment Routing based on MPLS (SR-MPLS), or Segment Routing based on IPv6 (SRv6) [2]. SFC-aware VNFs implement mechanisms to handle the SFC headers, i.e., their implementation is specific to the used SFC technology. However, it is desirable to use SFC-unaware VNFs that are not specific to any SFC technology.

SFC-unaware VNFs can be integrated into SF chains by coupling them with an SFC proxy [3]. Such an SFC proxy is applied per SF within a SF chain. It receives SFC-encapsulated packets, removes their SFC header, sends them decapsulated to the VNF, and adds the SFC header again to the packets after they have been processed by the VNF. Most SFC proxies that are discussed in literature and Internet standards today work on the level of SF chains. They are either configured with information about the supported SF chain (static proxy) or learn relevant information from arriving packets (dynamic proxy). With the help of that information, packets returning from a VNF are classified by the SFC proxy and equipped again with a correct SFC header. SFC proxies working on the level of SF chains reach their limit when packets belonging to the same SF chain have different SFC headers. This happens, e.g., when per-packet metadata is added to the SFC header. Examples are in-band network telemetry (INT) [4] or proof of transit [5]. Another use case where these proxies reach their limit is when load balancing between several SF instances is performed, resulting in different SFC headers for the same SF chain. Such use cases require more flexible SFC proxies that can handle SFC headers that may be different for each packet of a SF chain. Moreover, SFC proxies working on the level of SF chains are unable to support multiple SF chains with the same VNF.

Extended Berkeley Packet Filters (eBPF) [6] is a technology to execute simple programs inside an operating system kernel like Linux in a secure way. It can be used to apply operations like filtering or modifications to network packets that are processed by the kernel. This allows to add new functionality to the kernel.

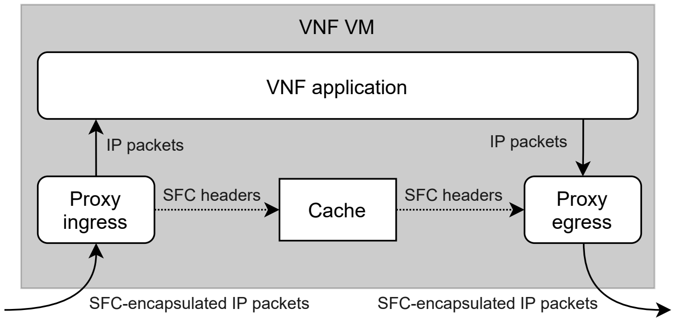

Fig. 1. Operation principle of a caching SFC proxy.

In this paper, we propose a caching SFC proxy that works on a per-packet level instead of on the level of SF chains. Its operation principle is shown in Figure 1. The caching SFC proxy removes the SFC header of an arriving packet, caches

This work was supported by the bwNET2020+ project which is funded by the Ministry of Science, Research and the Arts Baden-Württemberg (MWK). The authors alone are responsible for the content of this paper.

that header, and adds the cached header to the packet after processing by the VNF.

The proposed caching proxy implements two modes that differ in the keys used for caching. The first mode utilizes immutable parts of the packets' IP and TCP header as key. The second mode adds a unique identifier to the kernel metadata of the packets and utilizes it for caching. This approach is more versatile as it copes with VNFs modifying a packet's IP and TCP header, but it requires VNFs running in kernel mode. We show that these approaches are general enough to be applied to most traffic types on the Internet. We provide an eBPF-based implementation of the caching proxy and publish it on GitHub [7]. It supports any Linux-based system and achieves high throughput.

The rest of the paper is structured as follows. In Section II, we give a detailed introduction to SFC and eBPF. Then, we discuss related work in Section III. Section IV describes the concept of the proposed caching proxy, illustrates use cases, and gives details on a proof-of-concept implementation. We evaluate the performance of the prototype in Section V. Finally, we draw conclusions in Section VI and give an outlook on future work.

II. Technical Background

We give a short introduction to SFC and eBPF.

A. Service Function Chaining

Before SFC, providing a network service consisting of multiple chained NFs was a challenge. Network operators had to manually set up a series of physical appliances and manage the forwarding of the packets such that they traverse the desired NFs. With the advent of NFV [8] and SDN [9], provisioning of network services became more dynamic, scalable and manageable. The process of chaining VNFs requires configuration, deployment, and interconnection of the respective VNF instances, which are called SFs in the context of SFC. The resulting ordered set of SFs is called a SF chain. A reference architecture for SFC has been proposed by the IETF in RFC 7665 [3].

In this architecture, traffic is classified at the edge of an SFC-enabled domain. A classifier processes the traffic and decides based on configured policies and header information in the packets which SF chain should be traversed by each packet. After classification, the packet is encapsulated with an SFC header that contains information about the classification outcome. Within an SFC-enabled domain, SF Forwarders (SFFs) process steering information in the SFC encapsulation and forward traffic accordingly. For SFC-unaware NFs, SFC proxies are necessary to remove the SFC header before the packet can be further processed, and to add the encapsulation again after successful processing by the SF. An egress node processes traffic leaving the SFC-enabled domain and removes any SFC-related information from the packets not to leak sensitive data outside the SFC domain. The specific implementation of the SFC-encapsulation may be realized in different ways, e.g., via Network Service Header (NSH) [1], Segment Routing (SR) based on MPLS (SR-MPLS) [2], or SR based on IPv6 (SRv6) [2].

With NSH, the classification outcome is encoded in a Service Path Identifier (SPI) field of the NSH to indicate the used SF Path (SFP). A Service Index (SI) field indicates which SF should be traversed next. This approach requires the forwarding elements to be NSH-aware and to be reconfigured appropriately for every new SFC.

SR implements source routing. In SR-MPLS a header stack is used for that purpose, SRv6 utilizes a segment list. When SR-MPLS or SRv6 are used for SFC, either the classifier encodes the path of a SF chain in the respective header, or another downstream node performs this task based on the classification outcome. These approaches have the benefit that the underlay network does not need to be reconfigured for every new SFC.

Additional metadata can also be carried in the SFC encapsulation, e.g., in NSH context headers or in special-purpose MPLS labels [10]. Examples include telemetry data, OAM data, or context data. This data may be used or augmented by forwarding elements such as the SFF or SFC proxy, or by SFC-aware SFs, e.g., in the case of in-band OAM.

SFC has use cases in various fields such as broadband networks, mobile networks, or datacenter networks [11]. In broadband networks, SFC may be used to dynamically and selectively provide services to users such as DPI, NAT, DS-Lite, NPTv6, or parental control, just to name a few. In mobile networks, SFC can help to ensure agreed service policies, to provide security and privacy functions, or to include other Value Added Services (VAS) [12]. In data center networks, SFC can improve server security by adding security SFs before the servers [13]. Additionally, in 5G network slicing, SFC plays a critical role in delivering sophisticated services per slice, and SFC can increase the network management flexibility in 5G networks [14]. Escolar et al. presented a scalable software SFF and classifier that supports network slicing and that is based on Open vSwitch (OVS) [15].

An SFC proxy facilitates integration of an SFC-unaware SF into a SF chain. A simple SFC proxy serves only a single SF chain and is configured with a header stack for that SF chain so that it cannot support SFC headers with packet-specific metadata. These are obvious drawbacks. However, there are challenges for more sophisticated proxies. In contrast to transparent SFs, opaque SFs alter packet headers. NAT is an example. Changed packet headers make it difficult to recognize packets from different SF chains after processing. Furthermore, SFs may drop packets or even generate new packets. The latter is a problem when SFs serve multiple SF chains, since it is not clear to which SF chain a newly generated packet belongs. Headers with packet-specific metadata require SFC proxies to cache SFC headers per packet and add them again after processing by the SF. Most proxies either support only transparent SFs or only a single SF chain. In Section V, we show that the proposed caching SFC proxy supports multiple SF chains, opaque in-kernel SFs, SFs that drop packets, and SFC headers carrying packet-specific metadata.

B. *eBPF*

Extended Berkeley Packet Filters (eBPF) support secure and isolated execution of custom programs within the Linux kernel. Originating back in 1992, the original purpose of the BPF technology was to write arbitrary user-level packet filters for networking tools like tcpdump. eBPF is an extension of this technology to allow for the execution of general-purpose programs, written in a subset of C. Using a compiler backend, e.g., for LLVM, the C code is translated into BPF bytecode, and can then be just-in-time (JIT) compiled to native code by the kernel. Thereby, the performance of eBPF programs is close to natively compiled in-kernel code.

Today, eBPF is considered a new type of software or technology that facilitates adding/altering functionality to/of the Linux kernel, without changing the kernel source code or writing kernel modules. This adds a programmable layer to the Linux kernel that enables a whole new flexibility and new use cases such as kernel tracing/debugging, or high-performance load balancing.

eBPF permits to reprogram the runtime behavior of the Linux kernel without compromising stability, safety, or execution efficiency. Thanks to a combination of sandboxing using an in-kernel VM, and strict verification of source code by the eBPF Verifier, it can be ensured that the execution of a program is safe. This includes guarantees that the code will terminate or that there is no access to arbitrary memory resources, possibly leaking sensitive information.

Compiled eBPF bytecode runs on different Linux systems, without having to recompile it. Also, (user space) implementations of compilers and interpreters are available that make it possible to run eBPF programs on different platforms and operating systems [16], [17], [18].

The execution of eBPF programs is event-driven. Different so-called hooks are available that eBPF programs can be attached to. Examples include system calls, tracepoints, or network events. The latter is particularly interesting for the present use case since programs can be triggered each time a packet is received or sent.

Since kernel version 4.1, eBPF programs can be attached to hooks within the Traffic Control (TC) subsystem on Linux. It offers features like shaping, scheduling, policing, dropping, classifying, or marking [19]. It may be configured using the TC utility which is part of the iproute2 package and which is installed by default on most Linux distributions. eBPF programs can be attached to TC qdiscs as filters or actions.

A special hook for eBPF is called XDP (eXpress Data Path). It is the earliest possible method to perform network packet processing, namely immediately after packets arrive at the network interface card (NIC). The network stack of the Linux kernel is not invoked and can be bypassed completely, allowing for high-performance network processing.

Storing state and sharing information can be done through eBPF maps. Those maps can be used to write and retrieve data, and are accessible by both in-kernel eBPF programs and user space applications. Different types of eBPF maps are available such as arrays, hash maps, or least recently used (LRU) hash maps.

eBPF programs also have write access to the socket buffer that stores the packets and corresponding metadata. This allows for attaching identifiers to the metadata of packets, which makes them uniquely identifiable even if header fields are modified during kernel processing. Once packets leave the kernel space and enter user space, this metadata is lost as the socket buffer is not involved anymore.

To conclude, TC filters and actions written in eBPF can support many use cases. As an example, eBPF has been used for creating programmable network functions based on SRv6 [20]. Also, eBPF has been used in the context of SFC for different purposes such as monitoring VNF packet processing time [21] or building flexible, low-latency, high-throughput VNFs [22]. Castanho et al. proposed a transparent SFC architecture that implements SFC in eBPF-based stages, without the need to adapt SFs or network devices [23]. An extensive survey with more detailed information about eBPF and XDP was presented by Vieira et al. [6].

III. Related Work

Various approaches for the integration of SFC-unaware SFs into SFC-enabled domains via proxies have been discussed at IETF and in academia. We review them in the following.

Song et al. [24] describe several mechanisms to map cached SFC headers to packets. In case of transparent SFs that do not modify the packet, they propose to add an ID that identifies the SFC header to the packet using either a VLAN tag, or VXLAN. This is not transparent to the SF and requires that it supports the respective technology. Alternatively, they propose to use the source MAC address or the 5-tuple of the packet for mapping. For opaque SFs, the SF needs to inform the SFC proxy about mapping rules via a SFC control plane. While these mechanisms use caching for the SFC headers, the key space is limited when using VLAN or VXLAN. Thus, it is not possible to cache SFC and metadata headers on a per-packet level. When using the 5-tuple or MAC adresses for mapping, packets belonging to the same flow cannot be kept apart at all.

Cabellos et al. [25] use the LISP Mapping System to store SFC headers in the form of the NSH. The SFC proxy stores the NSH data in the Mapping System using a map-register message. It uses the 5-tuple of the packet for indexing. After processing by the SF, the SFC proxy retrieves the NSH data using a map-request. Similar to Song et al., the 5-tuple is not sufficient for caching data on a per-packet level.

Clad et al. [2] describe both a static and a dynamic proxy for service programming using SR-MPLS and SRv6. Both proxies map the MPLS label stack or SRv6 segment list to a network interface of the proxy. Thus, the proxy cannot perform mapping on a per-packet level and a dedicated network interface is needed for each cache entry.

Ueno et al. [26] extend the proxy proposed by Clad et al. with a tagging mechanism. They use arguments that are encoded into SRv6 Segment Identifiers as a tag for caching the headers. This tag is then encoded either into the IPv4 ToS field, or the IPv6 TC field of the packet before sending it to the SF. Similar to the previous proxies, this does not enable mapping on a per-packet level.

srext [27]–[29] is a Linux kernel module that implements the SRv6 network programming model. It can be used as a proxy for SRv6-based SFC. However, each VNF may only be part of a single SF chain.

SRNK [30] is a proxy for SRv6-based SFC implemented in the Linux kernel. Here, VNFs are executed in Linux network namespaces and their network interface identifiers are used for mapping packets to SF chains. Information about the SF chain of a VNF is learned automatically. Hence, every VNF may only be part of one SF chain and caching of SFC and metadata headers cannot be done on a per-packet level.

Abdelsalam et al. [31] discuss implications of using SR-unaware applications in SR-based SFC and analyze issues of deploying SR-uaware applications as VNFs. They describe the operations performed by proxies as “complex” and “not efficient”.

IV. Concept and Implementation

In the following, we give an overview of the design of the caching proxy, followed by a description of two different operation modes and their implications on supported protocols. Furthermore, we give details on a proof-of-concept implementation.

A. Overview

The fundamental idea of the caching SFC proxy is to strip the SFC headers of an SFC-encapsulated packet and cache them together with a unique identifier of the packet. The proxy is designed to work with Linux-based VNFs and runs on the same system in the form of two eBPF programs, one for ingress processing and one for egress processing. It uses an eBPF map to cache the SFC headers.

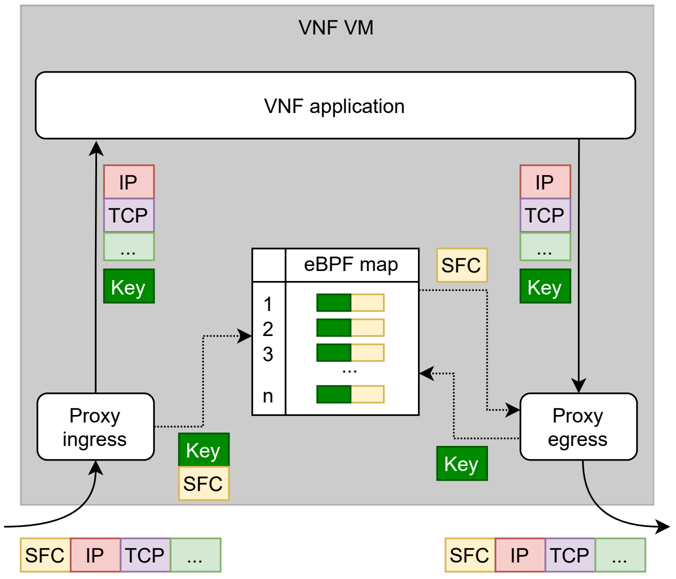

Fig. 2. Workflow of the caching SFC proxy.

The workflow of the caching SFC proxy is shown in Figure 2. When a packet containing SFC headers arrives at the VNF host, it is handed to the ingress proxy program. The ingress program computes a unique key that identifies the packet. How this key is computed depends on the type of VNF and is described in Section IV-B. The SFC headers are then stripped off the packet and stored in the eBPF map together with the identifier of the packet. The packet can then be processed by the VNF without its SFC headers. Afterwards, the packet is handed to the egress proxy program which retrieves and deletes the cached headers from the eBPF map using the previously generated unique identifier. In a final step, the egress proxy program adds the retrieved headers again to the packet.

B. Operation Modes

The caching SFC proxy supports two operation modes that differ in how cached headers are mapped to packets: one for kernel mode VNFs and one for user mode VNFs.

1) Operation for Kernel Mode VNFs: Once a packet arrives at the VNF machine, the ingress proxy program removes the SFC headers from packets and stores them in a hash table. It uses a sequence number as key and also attaches the same key to the packet as metadata. The metadata is stored in the `sk_buff` data structure in the Linux kernel which contains the control information for the packet [32]. The packet is then processed by the VNF program that is running in kernel mode. After processing of the packet by the VNF program is finished, an egress proxy program is executed. The egress proxy program uses the key that is stored in the packet’s metadata to fetch the corresponding SFC headers from the eBPF map. The SFC headers are then added to the packet again and the entry in the hash table is deleted.

2) Operation for User Mode VNFs: When packets leave the kernel space, their metadata in the `sk_buff` data structure are lost. Therefore, adding a sequence number to a packet’s metadata does not work so that the first operation mode is not feasible. Therefore, immutable parts of packet headers that are not removed by the VNF are used as key for mapping packets to cached SFC headers. As a consequence, this operation mode supports only protocols that use headers with sufficient information to clearly identify individual packets. For instance, the use of TCP/IP is already enough. Then, the source and destination IP addresses, the identification field of the IP header, the TCP source and destination port, and TCP sequence and acknowledgment numbers constitute a unique key to identify a packet. Another example are multimedia applications using RTP. Here, the combination of source and destination IP addresses, and timestamps uniquely identifies packets.

It is possible that hash collisions occur, i.e., two different keys produce the same hash values. To resolve such situations, the entire key, i.e., the header fields used to compute the hash are stored with the SFC headers in the eBPF map. Then the correct SFC header can be retrieved from the eBPF map.

As mentioned before, the key for this operation mode is taken from immutable parts of the header fields. If no such fields exist, this mode cannot be applied. An example is a NAT VNF running in user mode. For such kind of applications, the presented caching proxy is not applicable.

C. Other Use Cases

The caching mechanism for headers can be used to enable new use cases with legacy VNFs. This includes in-band network telemetry (INT), proof of transit (POT), and application-aware networking (APN).

INT [4] is used to collect telemetry data from the hops on the path of a packet, e.g., processing delay or queue occupancy. This information is added to the packet as additional metadata. If it is added using a header that the VNF cannot process, the header can be cached by the proxy similarly to SFC encapsulation headers. In addition, the proxy can be extended to directly support INT and add telemetry information to the packet, thus eliminating the need to implement INT in every VNF.

POT [5] is a mechanism based on Shamir's secret sharing scheme. It is used to prove that a packet was processed by a specific set of nodes. A central controller splits up a secret key into multiple shares and distributes them to the SFs of a SF chain. The SFs use their share of the key to update the POT data in each packet. A verifying node receives the full key from the controller and is located at the end of the chain. It uses the key and the POT data in the packet to verify that it traversed all SFs. Like with INT, implementing POT in the proxy is less work than adapting every VNF.

Application-aware networking (APN) [33] is a new technology to inform the network about requirements of specific applications. It achieves this by adding an attribute that expresses these requirements to packets. This attribute can be either encoded within headers like MPLS, VXLAN, and SRv6, or a dedicated APN header can be used [34]. The eBPF-based proxy can be adapted to cache these headers in order to make APN compatible with legacy VNFs that do not support the headers used for APN.

D. Prototypical Implementation

In the following, we give details on a prototypical implementation of an eBPF-based caching proxy for SFC with SR-MPLS. We describe the ingress and egress eBPF programs and how the MPLS headers are cached in an eBPF map.

a) Ingress Processing: Ingress processing starts with parsing the packet's IP, TCP, and MPLS headers. If no MPLS headers are present, ingress processing is stopped and the packet is forwarded without any alterations. If MPLS headers are present, the key to store the headers is generated. In case of a kernel mode VNF, a 64 bit sequence number is used as a key and added to the packet's metadata. In case of a user mode VNF, the IP and TCP headers are concatenated and all non-immutable fields (IP TTL, IP total length, IP checksum, and TCP checksum) are set to zero. Then, the MPLS headers are stored in an eBPF map using the previously generated key. If an entry with the same IP and TCP headers is already present in the map, the packet is dropped as it is not distinguishable from a packet that is already being processed by the VNF. At the end, the MPLS labels are removed from the packet.

The ingress processing is executed using XDP. Using XDP can result in a better performance compared to an eBPF-based TC filter as it is executed at an early step in the NIC driver and may be offloaded to the NIC itself. The performance of offloaded eBPF/XDP programs has been extensively evaluated by Hohlfeld et al. [35].

b) Egress Processing: Egress processing starts with parsing the packet's IP and TCP header. The key to retrieve the MPLS headers from the eBPF map is generated in the same way as in ingress processing. In the absence of a matching entry in the eBPF map, the packet is either forwarded as is or dropped depending on the configuration. Otherwise, the MPLS headers are retrieved from the matching entry and deleted from the eBPF map. Lastly, all but the topmost MPLS header that identifies the current VNF are pushed on the packet again.

In contrast to ingress processing, it is not possible to use XDP for egress processing as XDP is not implemented for the TX path in the Linux kernel. While experimental patches exist that implement XDP for the TX path of the Linux kernel, we chose not to use them as this would require using a modified kernel as this is not acceptable for production purposes. Instead, the egress processing is loaded as a TC filter.

c) Header Caching: Caching MPLS headers is implemented using an eBPF map as hash table. This type of data structure is not implemented by the eBPF programs itself, but by the Linux kernel. Hashing of the keys of the entries, in our case the immutable parts of the IP and TCP headers, is done by the kernel implementation of the eBPF map. The kernel implementation also stores the unhashed keys together with the values so that hash collisions can be resolved. Thus, the caching SFC proxy does not need custom handling of hash collisions.

Entries that are matched during egress processing are removed from the eBPF map. The map has a fixed size and uses LRU as replacement strategy when a new entry is added to a full data structure. If the map size is too small, entries of packets that are still in the queue of the VNF may be replaced, which causes packet drops. Therefore, the size of the map has to be chosen under consideration of the processing time and the throughput of the VNF.

V. Evaluation

In this section we evaluate the performance of the caching proxy and validate its functionality under various conditions. We describe the evaluation methodology before we conduct various experiments. First, we compare the maximum goodput of the caching proxy with the one of a static SFC proxy based on the Linux *mpls_router* module. Then, we show that the caching proxy copes well with packet loss and delay caused by a VNF. Finally, we demonstrate that the caching proxy supports opaque VNFs and multiple SF chains.

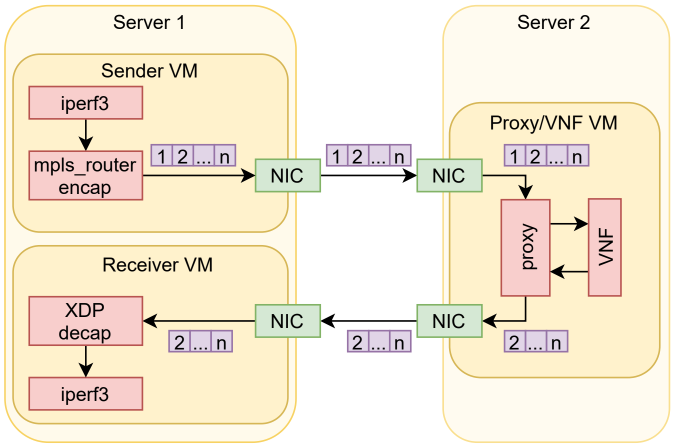

Fig. 3. Evaluation setup. Purple boxes represent the MPLS labels.

A. Evaluation Setup and Methodology

We describe the logical setup of the testbed and its hardware implementation. Then, we explain how experiments are conducted. Finally, we derive the number of parallel TCP flows needed to maximize the SFC goodput in a system with minimal complexity.

1) Logical Setup: The evaluation setup is shown in Figure 3. A sender virtual machine (VM) is connected via proxy/VNF VM to a receiver VM, setting up a SF chain. The sender pushes a set of MPLS labels on all outgoing packets using the *mpls_router* kernel module. The proxy/VNF VM runs a VNF application that is accessed either via a proxy or directly depending on the experiments. Traffic between sender and receiver is processed in both directions by the VNF application. The receiver VM removes all MPLS labels of the traffic. This operation is performed by a simple self-written XDP program for performance reasons that are explained in Section V-B2.

Due to the simplicity of the setup, MPLS label stacks in packets are not needed for traffic steering along the path. However, their presence is important as label pushing and popping is performed by ingress and egress processing of SFC proxies, which causes major effort.

2) Hardware Testbed: The 3 VMs run on 2 servers, each configured with an Intel(R) Xeon(R) Gold 6134 CPU running at 3.2 GHz, 128 GB of RAM, and 2 100G Mellanox ConnectX-5 network interface cards. The sender and receiver VMs are configured with 4 virtual CPU cores, 16 GB of RAM, and 1 ConnectX-5 NIC. The proxy/VNF VM is configured with 8 virtual CPU cores, 16 GB of RAM, and 2 ConnectX-5 NICs. The NICs are passed through to the VMs using SR-IOV. The NICs are connected to each other as shown in Figure 3. No overbooking is performed on the VM hosts and the virtual CPUs are pinned to physical cores to minimize the influence of virtualization on the experimental results. The VM hosts run Proxmox Virtual Environment 6.4, the VMs use Ubuntu 20.04. The parameters of the network stack, e.g., TCP parameters, are set to the default values of Ubuntu 20.04.

3) Experiment Design: The experiments determine the maximum goodput of a single SF chain. We call this metric SFC goodput. The traffic is sent by the sender, forwarded via the proxy/VNF, and received by the receiver. The experiments differ by various SFC proxy and VNF configurations.

To measure the SFC goodput, we leverage iperf 3.6 to determine a maximum TCP goodput. As a single TCP stream may not suffice to fully utilize the system, we run multiple TCP streams in parallel. For this purpose, iperf3 is executed multiple times in parallel instead of using the built-in parallel stream option to circumvent limited performance as iperf3 is a single-threaded application. In each run of an experiment, the overall goodput of the parallel iperf3 instances is computed over a duration of 10 minutes. We perform at least 10 runs per experiment and average their goodput values. We further derive confidence intervals for a confidence level of 95%.

We set the default capacity of the eBPF map to 10000 entries and the entry size to 10 MPLS labels. Likewise, the SFC encapsulation header contains 10 MPLS labels if not mentioned differently.

4) Maximizing SFC Goodput: We determine the number of parallel iperf3 instances needed to maximize the SFC goodput in a system with minimal complexity. To that end, we connect the sender via the VNF host to the receiver, but we bypass the SFC proxy and the VNF. We obtain $20\,\mathrm{Gb/s}$ for a single TCP stream and $60\,\mathrm{Gb/s}$ for 9 or more TCP streams. Therefore, we utilize 10 parallel iperf3 instances to determine goodput values in the following experiments.

B. Impact of SFC Proxy Variants

We evaluate the forwarding performance of a static SFC proxy and the caching SFC proxy, the latter with user and kernel mode VNFs due to the different operation modes. A major task performed by proxies is popping and pushing MPLS labels. Therefore, we investigate the forwarding performance depending on the size of the MPLS label stack. We consider only a single SF chain as the static SFC proxy can support only a single one. We utilize a forwarding-only VNF to minimize the delay by the VNF so that reduced forwarding performance can be mostly attributed to the SFC proxy.

1) Static SFC Proxy: We use the *mpls_router* module of the Linux kernel to configure a static SFC proxy on the VNF VM. It removes all MPLS labels of incoming packets and adds a static set of MPLS labels to all outgoing packets.

The SFC goodput for the static SFC proxy is given in Figure 4 for 0 to 10 MPLS labels in the SFC encapsulation header. It drops from approx. $42\,\mathrm{Gb/s}$ without any MPLS labels to approx. $15\,\mathrm{Gb/s}$ with 10 MPLS labels. This decrease in performance can be attributed to popping the labels during ingress processing. This hypothesis will be confirmed by further experiments in Section V-B2.

2) Caching Proxy with User Mode VNFs: The SFC goodput for the user mode SFC proxy is also indicated in Figure 4. It reaches approx. $33\,\mathrm{Gb/s}$ for 0-5 MPLS labels and decreases to approx. $29\,\mathrm{Gb/s}$ with more labels. The maximum size of the entries in the eBPF map is adapted to the number of MPLS

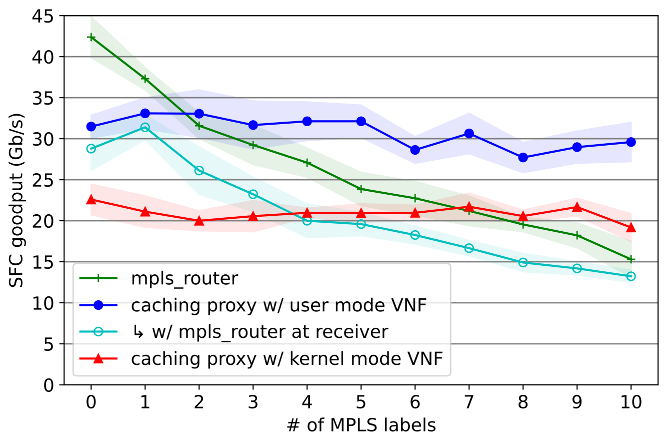

Fig. 4. SFC goodput for a forwarding-only VNF and different SFC proxies depending on the size of the MPLS label stack.

labels for each run. As the curve is rather flat, we conclude that the size of the entries has only little influence on performance.

For 0-2 MPLS labels, the SFC goodput for the *mpls_router* kernel module is larger than the one for the caching proxy with user mode VNFs. For more MPLS labels this is vice-versa. The performance of the *mpls_router* kernel module decreases with every additional MPLS label, while the caching proxy for user mode VNFs achieves almost equal goodput for any tested number of MPLS labels.

We show that the *mpls_router* module is rather inefficient in popping large label stacks. To that end, we substitute our self-written XDP decap program on the receiver VM through the *mpls_router* module. The curve for eBPF-based proxy with user mode VNF and *mpls_router* at receiver in Figure 4 shows lower SFC goodput than the corresponding curve with the XDP decap program on the receiver. Moreover, the curve has a similar shape as the one for the static SFC proxy based on the *mpls_router* module. We conclude that label popping in the *mpls_router* module is inefficient and significantly impacts the SFC goodput when used at the receiver or within a static SFC proxy. To avoid SFC goodput degradation in experiments due to inefficient popping of the label stack at the receiver, we utilize the XDP decap program by default. We also studied label pushing by the *mpls_router* module in a similar way, but there was no obvious bottleneck.

3) Caching Proxy with Kernel Mode VNFs: Figure 4 shows also the SFC goodput of the caching proxy with kernel mode VNF. It is approx. $10\,\mathrm{Gb/s}$ less than the goodput of the caching proxy with user mode VNF. This seems counter-intuitive at first sight, but can be explained by the different XDP execution paths used by the caching proxies for user and kernel mode VNFs.

In Figure 5 the execution paths of the caching proxy are shown for user mode and kernel mode VNFs. For user mode VNFs, the caching proxy is executed using the XDP capability of the Mellanox mlx5 driver in the Linux kernel. This means that the XDP program is executed at an early point while the packet is being processed by the driver of the NIC (see Figure 5 Path 1). The operation mode for kernel mode VNFs uses a driver-independent XDP implementation in the Linux kernel as the mlx5 driver does not support manipulation of metadata. In this case, the XDP program is not executed in the NIC driver, but at a later point while the packet is being processed by the ingress path of the Linux network stack (see Figure 5 Path 2). This enables the necessary manipulation of the packet metadata, but leads to an inevitable decrease in performance [36]. Drivers for other NICs, e.g., by Intel, support metadata but were not at our disposal. The execution path can be chosen when loading the proxy program. The implementation is the same in both cases.

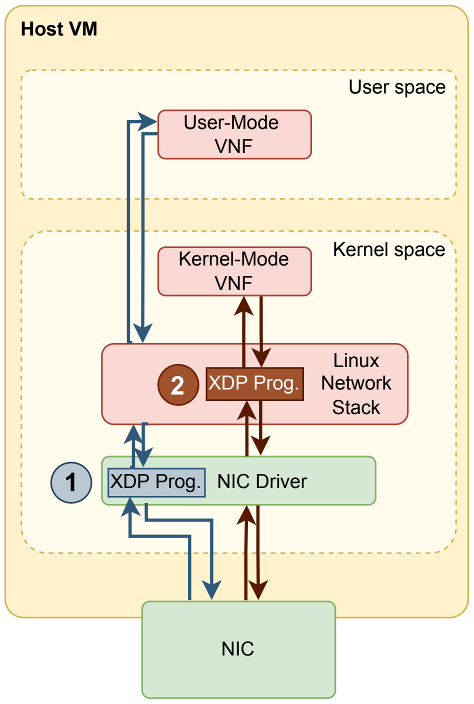

Fig. 5. Execution paths of XDP progams. Path 1 shows the driver path for user mode VNFs. Path 2 shows the generic path for kernel mode VNFs which facilitates manipulation of metadata.

C. Impact of Packet Drops by the VNF

We study the impact of packet loss on the operation of the caching SFC proxy. We set up a user mode VNF that uses iptables to randomly drop 1% of the packets. We monitor the SFC proxy and provide statistics in Table I.

Values of the counters *RX_IGNORED* and *TX_IGNORED* correspond to packets that are not handled by the SF chain. They are ignored by the proxy and forwarded without modification. ARP packets are an example. The number of packets that have been handled successfully during egress processing (*TX_SUCCESS*) is approx. 1% less than the corresponding number during ingress processing (*RX_SUCCESS*). Furthermore, there were no cache misses (*TX_NOT_CACHED*). Thus, VNFs that drop packets do not have any effect on the eBPF-based proxy.

TABLE I
CACHE STATISTICS FOR A USER MODE VNF THAT RANDOMLY DROPS 1% OF THE PACKETS.

Counter	Value
RX_IGNORED	71
RX_SUCCESS	373 494 929
TX_IGNORED	97
TX_NOT_CACHED	0
TX_SUCCESS	369 758 795

D. Impact of VNF Processing Time

The processing time of the VNF d together with the packet transmission rate r determines the average number of packets $n = d \cdot r$ that are in flight between ingress and egress processing of an SFC proxy (Little's Law). Many packets in flight are challenging for a caching proxy as it caches their headers in the hash map. If the hash map overflows, old entries are deleted due to the LRU cache replacement policy. When corresponding packets return from the VNF, they are dropped by egress processing due to missing headers in the hash map, which causes packet loss. This is not a severe problem as packet loss may also occur for other reasons, e.g., capacity shortage. However, it can reduce the SFC goodput significantly. We investigate this issue with a caching proxy and 10 MPLS labels. We use a delaying VNF in user mode that just delays the traffic for 5 ms. It is implemented with the TC NetworkEmulator.

We perform experiments with various cache sizes in terms of number of entries. The results are compiled in Table II. We test caches with 1000, 2500, 10 K, and 100 K entries. They lead to cache sizes between 97 kB and 9.7 MB. They all cause only little cache misses. However, loss rates of 0.04% are obviously large enough to significantly reduce the SFC goodput: while it is 4.96 Gb/s for 5000 entries, it is only 0.49 Gb/s for 1000 entries. This is caused by the underlying TCP protocol and the relatively long round-trip times due to the delaying VNF. The average occupancy of the hash table is the average number of packets in flight, which we computed with Little's Law. It is significantly lower than the cache sizes. Thus, a large safety margin is needed when the required cache size is calculated with Little's Law, the expected packet rate, and the VNF delay.

Generally, the SFC goodput here is much lower than in the experiments of Section V-B (~5 Gb/s vs. ~20 Gb/s). This is again caused by the delaying VNF that increases the round-trip time, which reduces the goodput of the TCP streams.

We further observe that a cache size of 5000 entries leads to the largest SFC goodput. Smaller caches cause more packet loss, which decreases SFC goodput. However, larger caches also decrease SFC goodput. We attribute this phenomenon to longer access times to the eBPF map for larger cache sizes. Thus, the size of the eBPF map should be chosen according to the expected number of packets in flight plus some generous safety margin. This value depends on the application and the maximum traffic load.

TABLE II
CACHE AND TRANSMISSION STATISTICS FOR A USER MODE VNF THAT DELAYS PACKETS BY 5MS.

Cache size	Cache misses	Transmitted packets / SFC goodput	Avg. number of packets in flight
1000 entries 97 kB	2.53 % 720 716	28 533 119 pkts 0.490 Gb/s	238 pkts
2500 entries 243 kB	0.04 % 70 027	168 452 033 pkts 2.894 Gb/s	1404 pkts
5000 entries 485 kB	0.01 % 39 727	284 786 870 pkts 4.959 Gb/s	2373 pkts
10000 entries 970 kB	<0.01 % 22 466	271 554 601 pkts 4.712 Gb/s	2263 pkts
100000 entries 9.7 MB	0.00 % 0	261 273 777 pkts 4.599 Gb/s	2177 pkts

E. Support for Opaque VNFs in Kernel Mode

We prove that the caching proxy can support opaque VNFs in kernel mode. Such VNFs change header fields so that packets cannot be recognized after VNF processing without additional metadata. However, the caching proxy for kernel mode VNFs utilizes such metadata for packet recognition.

We configure a source NAT for packets using iptables. We carry out the experiment, monitor goodput, and collect statistics for the proxy. Even though the NAT modifies the IP and TCP headers, no cache misses are recorded. Thus, the caching of headers using a key stored in the packet metadata works as intended. The goodput drops to approx. 17 Gb/s compared to 20.5 Gb/s for a forwarding-only VNF in Section V-B3.

F. Single VNF for Multiple Service Function Chains

In the previous sections we considered only a single service function chain. Now we provide two sender VMs and two receiver VMs. We connect each sender/receiver pair with 5 iperf3 streams each via a single forwarding-only VNF. They achieve together an overall SFC goodput of about 29.84 Gb/s for a user mode VNF and about 19.71 Gb/s for a kernel mode VNF. This experiment demonstrates that caching proxies can support a single VNF for multiple service function chains, which is unlike with static proxies.

VI. CONCLUSION

In this paper, we presented a service function chaining (SFC) proxy that is capable of caching SFC and metadata headers while a packet is processed by a virtual network function (VNF). The proxy uses either parts of the IP and TCP headers, or a sequence number that is stored as kernel metadata for the packet as a key for caching. The proposed proxy allows for caching SFC or metadata headers on a per-packet level instead of a per-SFC level like other proxies presented in IETF and by academia. This enhanced functionality enables use cases that require per-packet metadata like in-band network telemetry (INT) or proof of transit (POT). The implementation leverages eBPF and XDP for a high performance and for supporting VNFs on any Linux-based system.

We showed that the caching proxy outperforms a static proxy based on the *mpls_router* module when more than 2 labels are present in the SFC header. It copes well with VNFs dropping or delaying packets. We demonstrated that the proxy works with opaque network functions that modify packet headers, e.g., with network address translation (NAT). Finally, we showed that the caching proxy helps VNFs to support different service function chains at a time, which is unlike most other SFC proxies.

In future work, the caching SFC proxy may be adapted to IPv6, which is needed due to the lack of an identification field in the IPv6 header. The proxy may be extended to other protocols like RTP or IPsec. Its performance may be improved by NICs with XDP drivers supporting kernel metadata. Finally, applications based on SFC metadata, e.g., proof-of-transit (POT), in-band network telemetry (INT), or application-aware networking (APN) may be implemented.

References

[1] P. Quinn, U. Elzur, and C. Pignataro, "Network Service Header (NSH)," RFC 8300, January 2018. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8300

[2] F. Clad, X. Xu, C. Filsfils, D. Bernier, C. Li, B. Decraene, S. Ma, C. Yadlapalli, W. Henderickx, and S. Salsano, "Service Programming with Segment Routing," September 2021. [Online]. Available: https://www.ietf.org/archive/id/draft-ietf-spring-sr-service-programming-05.txt

[3] J. Halpern and C. Pignataro, "Service Function Chaining (SFC) Architecture," RFC 7665, October 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7665

[4] L. Tan, W. Su, W. Zhang, J. Lv, Z. Zhang, J. Miao, X. Liu, and N. Li, "In-band Network Telemetry: A Survey," *Computer Networks*, vol. 186, p. 107763, 2021.

[5] F. Brockners, S. Bhandari, T. Mizrahi, S. Dara, and S. Youell, "Proof of Transit," November 2020. [Online]. Available: https://www.ietf.org/internet-drafts/draft-ietf-sfc-proof-of-transit-08.txt

[6] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges and Applications," *ACM Computing Surveys*, vol. 53, no. 1, pp. 1–36, 2020.

[7] "eBPF SFC proxy repository on GitHub." [Online]. Available: https://github.com/uni-tue-kn/sfc-proxy

[8] B. Yi, X. Wang, K. Li, S. k. Das, and M. Huang, "A comprehensive survey of Network Function Virtualization," *Computer Networks*, vol. 133, pp. 212–262, 2018.

[9] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[10] Y. Liu, "Metadata in SR-MPLS Service Programming," September 2021. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-liu-spring-sr-sfc-metadata-01

[11] W. S. LIU, H. Li, O. Huang, M. Boucadair, N. Leymann, F. Qiao, Q. Sun, C. Pham, C. Huang, J. Zhu, and P. He.

[12] W. Haeffner, J. Napper, M. Stiemerling, D. Lopez, and J. Uttaro, "Service Function Chaining Use Cases in Mobile Networks," January 2019. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-sfc-use-case-mobility-09

[13] S. Kumar, M. Tufail, S. Majee, C. Captari, and S. Homma, "Service Function Chaining Use Cases In Data Centers," February 2017. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-sfc-dc-use-cases-06

[14] H. Hantouti, N. Benamar, M. Bagaa, and T. Taleb, "Symmetry-aware sfc framework for 5g networks," 2022.

[15] A. M. Escolar, J. M. A. Calero, and Q. Wang, "Scalable Software Switch Based Service Function Chaining for 5G Network Slicing," *IEEE International Conference on Communicaotions (ICC)*, pp. 1–6, 2020.

[16] "rbpf - A Rust (user-space) virtual machine for eBPF," retrieved: 2022-01-14. [Online]. Available: https://github.com/qmonnet/rbpf

[17] "generic-ebpf - A generic eBPF runtime." retrieved: 2022-01-14. [Online]. Available: https://github.com/generic-ebpf/generic-ebpf

[18] "uBPF - A Userspace eBPF VM," retrieved: 2022-01-14. [Online]. Available: https://github.com/iovisor/ubpf/

[19] "tc - show / manipulate traffic control settings." retrieved: 2022-01-14. [Online]. Available: https://manpages.debian.org/buster/iproute2/tc.8.en.html

[20] M. Xhonneux, F. Duchene, and O. Bonaventure, "Leveraging eBPF for Programmable Network Functions with IPv6 Segment Routing," *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pp. 67–72, 2018.

[21] N. Van Tu, J.-H. Yoo, and J. W.-K. Hong, "PPTMon: Real-Time and Fine-Grained Packet Processing Time Monitoring in Virtual Network Functions," *IEEE Transactions on Network and Service Management (TNSM)*, vol. 18, no. 4, pp. 4324–4336, 2021.

[22] ——, "Building Hybrid Virtual Network Functions with eXpress Data Path," *International Conference on Network and Services Management (CNSM)*, pp. 1–9, 2019.

[23] M. S. Castanho, C. K. Dominicini, M. Martinello, and M. A. Vieira, "Chaining-Box: A Transparent Service Function Chaining Architecture Leveraging BPF," *IEEE Transactions on Network and Service Management (TNSM)*, 2021.

[24] H. Song, J. You, L. Yong, Y. Jiang, L. Dunbar, N. Bouthors, and D. Dolson, "SFC Header Mapping for Legacy SF," September 2016. [Online]. Available: http://www.ietf.org/internet-drafts/draft-song-sfc-legacy-sf-mapping-08.txt

[25] A. Cabellos-Aparicio, S. Barkai, B. Perlman, V. Ermagan, F. Maino, and A. Rodriguez-Natal, "Map-Assisted SFC Proxy using LISP," October 2015. [Online]. Available: http://www.ietf.org/internet-drafts/draft-cabellos-sfc-map-assisted-proxy-00.txt

[26] Y. Ueno, R. Nakamura, and T. Kamata, "SRv6 Tagging proxy," October 2019. [Online]. Available: http://www.ietf.org/internet-drafts/draft-eden-srv6-tagging-proxy-00.txt

[27] A. Abdelsalam, F. Clad, C. Filsfils, S. Salsano, G. Siracusano, and L. Veltri, "Implementation of virtual network function chaining through segment routing in a linux-based NFV infrastructure," in *IEEE Conference on Network Softwarization (NetSoft)*, 2017, pp. 1–5.

[28] A. Abdelsalam, "Chaining of Segment Routing Aware and Unaware Service Functions." *IFIP-TC6 Networking Conference (Networking)*, pp. A3–A4, 2018.

[29] "SREXT - A Linux Kernel Module Implementing SRv6 Network Programming Model," retrieved: 2022-01-04. [Online]. Available: https://github.com/netgroup/SRv6-net-prog

[30] A. Mayer, S. Salsano, P. L. Ventre, A. Abdelsalam, L. Chiaraviglio, and C. Filsfils, "An Efficient Linux Kernel Implementation of Service Function Chaining for Legacy VNFs Based on IPv6 Segment Routing," in *IEEE Conference on Network Softwarization (NetSoft)*, 2019.

[31] A. Abdelsalam, S. Salsano, F. Clad, P. Camarillo, and C. Filsfils, "SERA: Segment Routing Aware Firewall for Service Function Chaining Scenarios," *IFIP-TC6 Networking Conference (Networking)*, pp. 46–54, 2018.

[32] "The Linux Foundation Wiki - networking:sk_buff," retrieved: 2021-12-15. [Online]. Available: https://wiki.linuxfoundation.org/networking/sk_buff

[33] Z. Li, S. Peng, D. Voyer, C. Li, P. Liu, C. Cao, G. Mishra, K. Ebisawa, S. Previdi, and J. Guichard, "Application-aware Networking (APN) Framework," October 2021. [Online]. Available: https://www.ietf.org/archive/id/draft-li-apn-framework-04.txt

[34] Z. Li and S. Peng, "Application-aware Networking (APN) Header," October 2021. [Online]. Available: https://www.ietf.org/archive/id/draft-li-apn-header-00.txt

[35] O. Hohlfeld, J. Krude, J. H. Reelfs, J. Rüth, and K. Wehrle, "Demystifying the Performance of XDP BPF," *IEEE Conference on Network Softwarization (NetSoft)*, pp. 208–212, 2019.

[36] "Cilium: BPF and XDP Reference Guide," retrieved: 2022-01-02. [Online]. Available: https://docs.cilium.io/en/v1.11/bpf/