Integrating Netfilter into SRv6 Routing Infrastructure of Linux as an SR-Aware Network Function

Kaito Sawada
Keio University
Kanagawa, Japan
sabaniki@wide.ad.jp

Ryo Nakamura
The University of Tokyo
Tokyo, Japan
upa@nc.u-tokyo.ac.jp

Keisuke Uehara
Keio University
Kanagawa, Japan
kei@wide.ad.jp

Abstract—**This paper proposes a new SRv6 End behavior, called End.AN.NF, integrating Linux netfilter as a network function for service function chaining by Segment Routing (SR). End.AN.NF allows netfilter-based applications to be executed as SR-Aware applications without modification, as it applies netfilter to inner packets encapsulated in SRv6 while performing the basic SRv6 End behavior. Furthermore, End.AN.NF utilizes the argument of the segment identifiers to mark packets. Consequently, this enables netfilter-based applications to match the marks on packet buffers and change rules to be applied. We implemented End.AN.NF on the Linux kernel and evaluated its performance. The evaluation shows that End.AN.NF achieves 27% higher throughput and 3.0 microseconds lower latency than applying netfilter to SRv6-encapsulated inner packets by End.DT4 and H.Encaps.**

Index Terms—**Service Function Chaining, Segment Routing, SRv6**

I. Introduction

Service Function Chaining (SFC) is a topic that has been studied in contexts of Software Defined Network (SDN) and Network Function Virtualization (NFV) [1]–[4]. SFC steers packets to Network Functions (NFs) according to predetermined rules about the order and type of NFs to traverse. SDN controllers or routing protocols install the rules to steer packets, and the headers of packets may include information matched by these rules. In SFC, routers require to select the next hops for forwarding packets regardless of the shortest path. SFC offers flexible and economical alternatives to today's static environments for Cloud Service providers (CSPs), Application Service Providers (ASPs), and Internet Service Providers (ISPs) [5].

Several technology candidates could achieve SFC, such as OpenFlow [6], Network Service Header (NSH) [7], MPLS [8] and so on. These technologies can steer packets to intended NFs based on rules, regardless of the shortest path. In OpenFlow, a controller installs flow rules to OpenFlow switches explicitly. OpenFlow switches forward packets to go through intended NFs with properly managed match the rules. This architecture allows flexible path control that is not based on traditional routing protocols. NSH identifies an NF by Service Path Identifier (SPI) and Service Index (SI). NSH nodes forward packets according to SPI and SI by encapsulation. NSH creates a dedicated overlay network called service plane and allows service forwarding to occur within that plane without modifying the underlying network topology. In MPLS, instead of using NSH directly, MPLS label stack contains the hop by hop order to pass through, routers and NFs. This approach also achieves steering packets for SFC without modifying the underlying network topology.

Segment Routing (SR), especially Segment Routing over IPv6 (SRv6), is also one of the technologies used for implementing SFC. SR represents every entity, such as links, nodes, and services, in a network by *segments*. A header of a packet contains a list of the segments. The list of segments indicates a path that the packet should pass through. SRv6 uses an IPv6 address as an identifier for a segment, called Segment Identifier (SID). In other words, SRv6 leverages IPv6 routing infrastructure as its underlay to deliver packets through arbitrary order of segments. SRv6 achieves SFC by assigning segments to the NFs to be performed on those segments, and forwarding packets accordingly.

In SRv6, NFs represented by SIDs would achieve SRv6-based SFC. However, it is not obvious how to integrate the behavior of NFs above the SRv6 layer and the underlying IPv6 routing infrastructure. For example, let us consider a Network Address Translation (NAT) for IPv4 packets as an NF in an SRv6 network. IPv4 packets are encapsulated in outer IPv6 headers involving SR Header (SRH). If the NAT implementation is not aware of SRv6, it requires SR-proxies [9] for SR-unaware NFs that introduce additional complexity [10]. If the implementation can perform both NAT for inner packets and the SRv6 forwarding behavior simultaneously, it would be layer violation. Such an implementation in Linux, SERA [11], acts as an End behavior by modified iptables actions, although the Linux kernel has the capability of performing an End behavior in its IPv6 routing and forwarding infrastructure.

Along these lines, in this paper, we introduce an SR-Aware service function, called `End.AN.NF`, which enables existing netfilter-based applications to be SR-Aware applications without modification. More specifically, it integrates Linux netfilter as an NF while leveraging the IPv6 routing infrastructure of Linux. We designed End.AN.NF to treat netfilter hook points transparent to the SRv6 inner packet. We implemented End.AN.NF on the Linux kernel and evaluated its throughput and latency. The evaluation shows that End.AN.NF achieves 27% higher throughput and 3.0 microseconds lower latency than applying netfilter to SRv6-encapsulated inner packets by the combination of End.DT4 and H.Encaps. In addition, the latency of End.AN.NF is the same as that of End behavior in microsecond resolution.

II. Background and Related Work

SRv6 [12] is one of the source routing architectures that uses IPv6. It allows network operators or applications to specify intermediate points that a packet passes through by embedding a sequence of identifiers into the IPv6 extension header, called SRv6 header (SRH) [13]. The identifiers are called by Segment Identifiers (SIDs), and each SID represents a specific function to be performed at a specific location in a network. To specify which SID is the current SID in the SID list, SRH has a field called Segments Left (segleft). segleft is an index for the SID list, starting from $(number\ of\ SIDs) - 1$ and ending at zero. A SID is an IPv6 address associated with a specific segment in a network. The SID is structured as LOC:FUNCT:ARG, where LOC represents a locator, FUNCT is an identification of a local behavior associated with the SID, and ARG may encode additional information required for an action. The locator may also be represented as B:N, where B is the SRv6 SID block (an IPv6 prefix allocated for SRv6 SIDs) and N is the identifier of the node instantiating the SID.

When SRv6 node receives a packet whose destination IPv6 address is a local SID configured in the node, the SRv6 node performs pre-defined behavior associated with the SID. In the SRv6 context, the behaviors that SRv6 nodes perform are called End behaviors. Currently, RFC8986 [12] defines 15 types of End behaviors, and the most basic End behavior is End. End decrements the segleft of SRH of a received packet and replaces the destination IPv6 address with the next SID. Next, the SRv6 node forwards the packet to the next hop in accordance with the updated destination IPv6 address. RFC8986 also defines behaviors that encapsulate packets in SRH involving SID lists, which are called Headend behaviors.

SIDs in SRv6 are IPv6 addresses; therefore, advertising SIDs over the existing routing protocols can build a network based on SRv6. For example, H.Encaps and End.DT4, which are Headend and End behaviors respectively, can compose layer-3 VPN [14]. H.Encaps encapsulates IPv4 or IPv6 packets in outer IPv6 headers with SRH, and End.DT4 decapsulates the packets encapsulated in the SRH. When H.Encaps in an ingress SRv6 node encapsulates packets in outer IPv6 headers, whose destination address is End.DT4 SID of an egress SRv6 node, the encapsulated packets are forwarded to the egress SRv6 node in accordance with the LOC of the End.DT4 SID over the underlying SRv6 routing infrastructure. The egress SRv6 node receives the packets and performs End.DT4; the packets are then decapsulated and the inner packets are routed based on a Virtual Routing and Forwarding (VRF) table associating the ARG of the SID.

The above examples illustrate the behaviors associated with SIDs. On the other hand, some behaviors are not limited to encapsulation and decapsulation; NFs applied to packets can be also represented by SIDs. When an NF applies some network services for transit packets while performing End—decrementing segleft and updating destination IPv6 address with the next SID, such an NF is called an SR-Aware function. SERA [11] is an implementation of an SR-Aware function, integrated with Linux iptables. SERA extends Linux iptables so that iptables matches fields in SRH with iptables rules, to apply filtering rules for firewall purposes. SERA can also perform like End, forwarding packets toward the next SID. This design choice makes it difficult to integrate the SIDs associated with SERA into routing infrastructures; these SIDs in iptables cannot be advertised via the existing routing protocols, unlike the example of layer-3 VPN. As outlined above, how to integrate a form of NF and the underlying IPv6 routing infrastructure still has room for consideration.

In contrast to SR-Aware functions, various methodologies to integrate traditional, SR-unaware NFs into SRv6-based SFC have been proposed. SR Proxy [9] is the key component to connect SR-unaware NFs with underlying SRv6 networks. An SR Proxy receives packets destined to a local SID, passes inner packets without the SRH to a associated NF, attaches proper SRH to packets returned from the NF, and forwards the packets to tne next SID. Some SR Proxy implementations in Linux have been proposed [15]–[17]. On the other hand, SR Proxies fundamentally introduce additional complexities to networks. For example, SR proxies need to determine proper SID lists attached to packets returned from NFs. There is a possibility that inner packets have arbitrary destinations and arbitrary sources. Thus, SID lists that the SR proxy should attach can vary depending on the inner packets. SR proxies need to implement their own mechanism to determine proper the SID lists to be attached, for example, attaching static ones (End.AS) or caching some states inside proxy implementations [15]. Moreover, some issues for deploying SR proxies are addressed—a type of service that cannot co-exist with a specific SR Proxy type, service liveness detection, and an SID advertisement issue for services behind SR proxies [10].

III. Design and Implementation of End.AN.NF

As a new implementation of SR-Aware NF, we introduce End.AN.NF, which integrates the ability to filter and mangle packets by netfilter into the routing infrastructure in Linux. End.AN.NF means an End behavior of SR-Aware Native function for NetFilter. We also designed End.AN.NF to leverage the IPv6 routing stack of the Linux kernel. An End.AN.NF SID is represented as an IPv6 routing table entry; therefore, its route can be exported and advertised to other nodes via the existing routing protocols and their implementations transparently. Moreover, End.AN.NF applies netfilter rules to inner packets encapsulated in SRv6. This enables the use of any netfilter-based applications, such as selective packet discarding and applying NAT for traffic, configured via nftables [18] and iptables [19], as SR-Aware NFs without modifying their implementations.

netfilter has three hook points in Linux layer-3 packet forwarding flow, which applies netfilter rules to packets at different timings: prerouting, forward, and postrouting. Figure 1 depicts a flow of transit packets and netfilter hooks to be applied. As shown, there are two stages for applying netfilter hooks to an SRv6 encapsulated packet as an IPv6 packet including the SRH and its inner packet without the SRH. First, when a Linux-based SRv6 nodes that implements `End.AN.NF` receives an IPv6 packet, its kernel applies the prerouting hook to the received packet, and then performs longest prefix matching for the destination IPv6 address as usual. If the destination address is a local `End.AN.NF` SID, the kernel passes the packet to the `End.AN.NF` implementation, otherwise the kernel forwards the IPv6 packet to a corresponding next hop while applying forward and postrouting hooks. Meanwhile, `End.AN.NF` applies prerouting, forward, and postrouting hooks again, but to the inner packet encapsulated in the SRH. During the netfilter applying in the stage of `End.AN.NF`, the SRH is hidden by `End.AN.NF`, so netfilter does not have to consider to treat the SRH.After `End.AN.NF` finishes, the destination address of the outer IPv6 header is replaced with the next SID, and the encapsulated packet returns to the usual forwarding path.

`End.AN.NF` utilizes the `ARG` field in the SID to mark packets. `ARG` is the lower bits of a SID [12]. The SRv6 specification allows End behaviors to utilize `ARGs` in accordance with their specific behaviors. In `End.AN.NF`, `ARG` of SIDs is attached to packet buffers as a mark. netfilter-based applications can match the marks on packet buffers and change rules to be applied. Therefore, operators can adjust rules for traffic based on `ARG` even behind a single `End.AN.NF` SID.

Algorithm 1 describes how `End.AN.NF` passes a packet to a netfilter hook point. First, `End.AN.NF` extracts the `ARG` value from the destination address of a received packet if the `ARG` length is specified for this `End.AN.NF` SID. The extracted `ARG` value is attached to the packet buffer as a mark. Next, `End.AN.NF` switches the head of the packet buffer from the outer SRH to the inner packet, and pass the buffer to a netfilter hook. After rules installed in the hook are applied to the inner packet, `End.AN.NF` restores the head of the packet buffer from the inner packet to the outer SRH, and takes the packet to the next process. This procedure occurs three times for each hook point illustrated as the red rectangles in Figure 1.

The Linux kernel implements End behaviors as routing table entries whose destinations are SIDs of the functions. This mechanism is called `seg6local`. `End.AN.NF` is one of the End behaviors, so its implementation also leverages `seg6local`. As shown in the Figure 2, we can confirm that the kernel treats the SID representing `End.AN.NF` as a routing table entry similar to the others End behaviors. When routing software or iproute2 adds the SID as a routing table entry, it is possible to advertise the routes in the kernel routing table using the traditional routing protocols. We con-

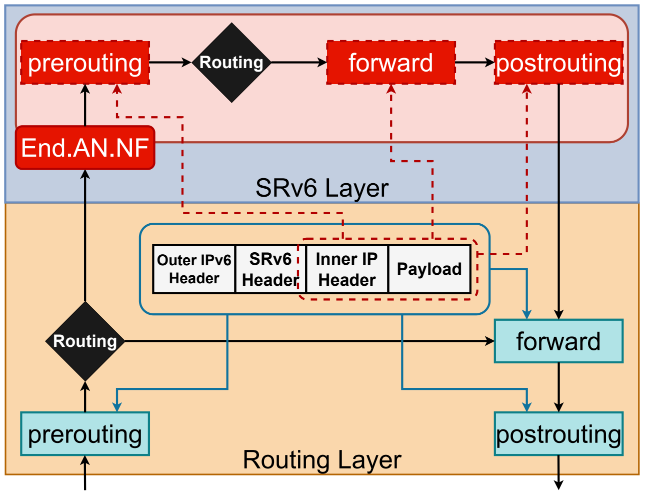

Fig. 1. `End.AN.NF` applies three netfilter hook points, prerouting, forward, and postrouting, to inner packets encapsulated in SRv6.

firmed that FRRouting [20] can advertise SIDs associated with `End.AN.NF` in the kernel as IPv6 routes to other routers via BGP. The architecture of `End.AN.NF` is highly compatible with the existing NFs because it can use the existing routing protocols for routing control. This architecture is one of the ways to realize SR-Aware NF using Linux netfilter.

IV. Evaluation

We conducted three experiments to evaluate the performance of our implementation: two focused on throughput and one on latency. The first experiment examines throughput based on the packet size, while the second assesses throughput in relation to the number of filter rules in a netfilter-based application. The third experiment investigates the latency associated with different packet forwarding mechanisms.

We compare the performance of `End.AN.NF` with three forwarding mechanisms: `End`, the combination of `End.DT4` and `H.Encaps`, and IPv4, which serves as a baseline. As shown in Figure 1, when `End.AN.NF` operates, a received packet passes through twice the number of hook points compared to `End`. Hence, the performance of `End.AN.NF` might be inferior to `End`. On the other hand, the performance of `End.AN.NF` is expected to be higher than the combination of `End.DT4` and `H.Encaps`. When applying netfilter rules to packets encapsulated in SRv6, a practical approach in a vanilla Linux kernel is the combination of `End.DT4` and `H.Encaps`. Vanilla Linux kernel does not have a method to apply netfilter to a packet that is still encapsulated in SRv6. Therefore, it needs to decapsulate the packet once and encapsulate it instead of the originally attached SRH again. A packet decapsulated by `End.DT4` passes through netfilter hook points as an IPv4 packet, and `H.Encaps` stores new SRH. Thus, this method has overheads because it requires `End.DT4` to decapsulate the packet and then `H.Encaps` to encapsulate the packet again. Therefore, it is anticipated that this overhead will lead to a degradation in performance.

Algorithm 1 Pseudo code of passing a packet to a netfilter hook point in `End.AN.NF`

```
1: function PassPacketToHook(packet)
2:     if the length of $ARG$ is specified for this End.AN.NF SID then
3:         Extract the $ARG$ value from the destination address of outer SRH
4:         Mark the $ARG$ value on the packet buffer $packet$
5:     end if
6:     Switch the head of packet buffer $packet$ from the outer SRH to the inner packet
7:     Pass $packet$ to a netfilter hook
8:     Switch the head of packet buffer $packet$ from the inner packet to the outer SRH
9: end function
```

```
$ ip -6 route | grep End
2001:db8:1::/96  encap seg6local action End.AN.NF arglen 32 dev eth0 metric 1024 pref medium
2001:db8:2::200  encap seg6local action End dev eth0 metric 1024 pref medium
2001:db8:3::300  encap seg6local action End.DX4 nh4 192.168.99.1 dev eth1 metric 1024 pref medium
```

Fig. 2. The modified Linux kernel treats an `End.AN.NF` SID as an IPv6 routing table entry. We can manage the `End.AN.NF` routes with the existing tools such as iproute2.

The environment is the same configuration for all experiments. We prepared two machines directly connected with a 100 Gbps link. The two machines are identical: Intel(R) Xeon(R) Silver 4310 12-core CPU x2, 64-GB DDR4-2666 memory, and Intel E810 100 Gbps NIC. We disabled The CPU's hyperthreading function. One is used as a traffic generator, and the other as a System Under Test (SUT). In the traffic generator machine, we installed Ubuntu 22.04 and TRex [21], which we used to generate test traffic. In the SUT machine, we installed the customized Linux kernel 5.15.106, where we have implemented `End.AN.NF` and the customized iproute2 command to configure `End.AN.NF` SIDs. We also prepared two VLANs on the link between the two machines.

A. Throughput per packet size

We measured the throughput of `End.AN.NF`, `End`, IPv4, and the combination of `End.DT4` and `H.Encaps` while changing the packet size to increase. This experiment clarifies the change in throughput with packet size for each packet forwarding mechanism. We did not use any netfilter rules in this experiment. We assessed the throughput reduction of `End.AN.NF` in comparison to `End` and the performance improvement of `End.AN.NF` over the combination of `End.DT4` and `H.Encaps`.

We sent traffic generated by TRex on the traffic generator machine to the SUT machine, from a minimum packet length of 126 bytes to a maximum packet length of 1518 bytes. We calculated the packet length at the time of measurement as follows: $l = 174n + 126$. l means the length of the packet, and n means the number of measurements. We collected a total of 10 measurements, from $n = 0$ to $n = 10$.

The reason we chose a packet length of 126 bytes as a minimum packet length is the minimum length for a UDP packet with a tagged VLAN when the SID list length is two. `End.AN.NF` mandates a SID list length of at least two, as it decrements the packet's segleft, which must remain zero or above. With a SID list length of one, the segleft starts at zero, and decrementing it with `End.AN.NF` would result in a negative value. Conversely, `End.DT4` necessitates that the segleft is zero. In the measurement of the combination of `End.DT4` and `H.Encaps`, TRex generated packets with a SID list length of two but set the segleft to zero. To effectively use the Receive Side Scheduling (RSS) mechanism, TRex incremented both the destination and source addresses of the inner IPv4 packet during packet generation. For IPv4, we embedded dummy data in the UDP payload to match the SRv6 packet length, starting the packet length at 126 bytes. We also incremented the destination and source addresses during packet generation to leverage RSS effectively. Given that the maximum size of an Ether frame, including the tagged VLAN header, is 1518 bytes, we set the upper packet size limit to 1518 bytes for this measurement.

Figure 3 shows the result of this experiment. The throughput of `End.AN.NF` never degrades by more than 6% compared to `End` across all packet lengths. For a packet length of 1518 bytes, `End.AN.NF` exhibits the least throughput degradation relative to `End`, approximately 1.7%. Conversely, about 5.6% of the highest degradation occurs at a packet length of 478 bytes when comparing `End.AN.NF` to `End`. There is no correlation between packet length and throughput, which varied significantly. This degradation in throughput can be attributed to packets in `End.AN.NF` traversing twice the number of netfilter hook points compared to those in `End`, which remains within an acceptable range.

When comparing the throughput of `End.AN.NF` to the combination of `End.DT4` and `H.Encaps`, `End.AN.NF` consistently outperforms, irrespective of the packet length, as anticipated. Specifically, `End.AN.NF` achieves a throughput that is 26.7% higher than that of the `End.DT4` and `H.Encaps` combination. The throughput disparity between `End.AN.NF` and the combination of `End.DT4` and `H.Encaps` is influenced by packet length: shorter packets result in a larger relative performance gap, while longer packets yield a nar-

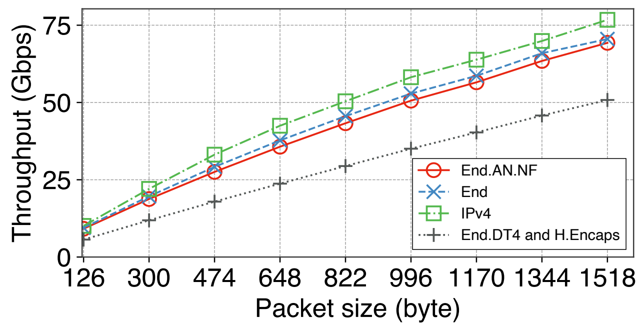

Fig. 3. Throughput per SRv6 End behaviors and IPv4

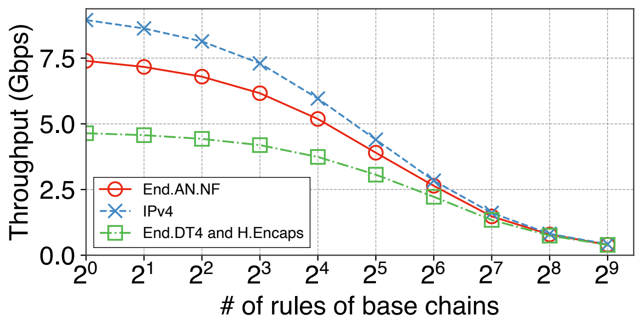

Fig. 4. Throughput per number of rules of base chains

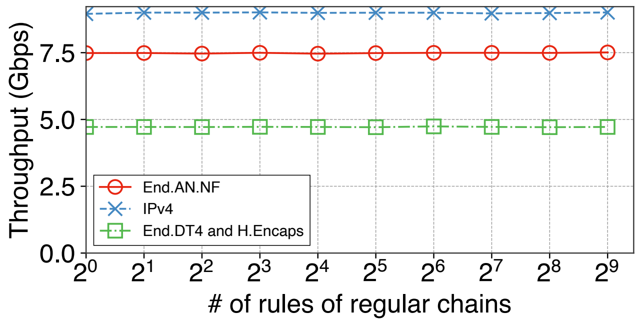

Fig. 5. Throughput per number of rules of regular chains

rower difference. The packet per second (pps) rate increases as packet size decreases. As a result, smaller packet sizes make any overheads in packet forwarding more noticeable.

B. *Throughput per number of filter rules installed in netfilter*

Next, we evaluated the throughput of `End.AN.NF`, IPv4, and the combination of `End.DT4` and `H.Encaps`, varying the number of filter rules installed in netfilter. We used nftables as a netfilter-based application to install filter rules. In nftables, the rules are expressed as sets of chains. There are two types of chains: base chains and regular chains. All filter rules in base chains apply to forwarding packets. nftables uses the regular chains only when the other chain refers to the regular chain. In our experiment, we gauged the throughput for each chain type, incrementing the count of each chain type. We anticipated a throughput decline with the addition of filter rules, irrespective of the forwarding mechanisms. This experiment aims to clarify the characteristic of the throughput degradation for each packet forwarding mechanism due to the filter rules.

We sent traffic generated by TRex on the traffic generator machine to the SUT machine. For this measurement, we set the packet length consistently at 126 bytes. Our choice for a 126-byte packet length aligns with the rationale provided in Section IV-A, which corresponds to the minimum length of a UDP packet with a tagged VLAN when the SID list length is two.

Figure 4 illustrates the throughput per number of rules of base chains. The chain rule is one of the worst cases of the nftables chain rule. In this measurement, we installed filter rules at the netfilter's forward hook point. These rules are consistently designed to accept all passing packets and then apply the same subsequent rule. netfilter permits the installation of multiple rules on a single hook point. Throughout the experiment, we incrementally increased the number of these identical cascading rules applied at this hook point. Across all packet-forwarding mechanisms, throughput diminishes with an increasing number of rules. As rule count rises, the throughput for all three mechanisms converges to approximately 0.4 Mbps. When comparing the throughput of `End.AN.NF` and IPv4, there is no pronounced disparity of characteristics in throughput decline, with `End.AN.NF` not showing any significant disadvantage relative to IPv4. Consistently, `End.AN.NF` outperforms the combination of `End.DT4` and `H.Encaps` in terms of throughput. However, this performance gap narrows with increasing rule count, reaching a mere 9% difference at 128 rules. As the rule count in the regular chain escalates, the advantage of `End.AN.NF` over the combination of `End.DT4` and `H.Encaps` diminishes.

Figure 5 presents the throughput as per number of rules in regular chains. Notably, there's no observed degradation in throughput, `End.AN.NF` consistently outperforms the combination of `End.DT4` and `H.Encaps`. The filter rules for regular chains were the same configuration as when we measured for the base chains, consistently designed to accept all passing packets and then apply the same subsequent rule. However, in this chain rule configuration, none of the defined regular chains are referenced by other chains, so none of the rules configured in regular chains apply to the packets. As a result, the actual number of rules applied as packets traverse the netfilter hook point remains unchanged.

C. *Latency of End.AN.NF, End, IPv4, and the combination of End.DT4 and H.Encaps*

We also measured the latency of `End.AN.NF`, `End`, IPv4, and the combination of `End.DT4` and `H.Encaps`. This experiment focused on latency. Our objective was to evaluate the latency of `End.AN.NF` by comparing its degradation relative to `End` and its improvement when juxtaposed with the combination of `End.DT4` and `H.Encaps`. For this evaluation, we used IPv4 latency as the baseline reference.

We measured packet forwarding latency using TRex. TRex can capture the time interval between packet transmission and

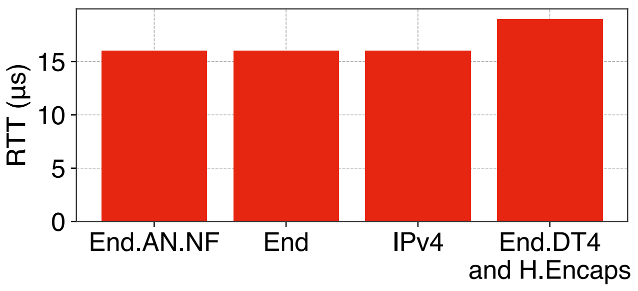

Fig. 6. Latency per SRv6 End behaviors and IPv4

reception in microsecond resolution. In this measurement, the packet length was 142 bytes: 126 bytes is the minimum packet length `End.AN.NF` requires, as explained in Section IV-A, and the next 16 bytes is used to pass information needed to measure latency by TRex. During the experiment, the traffic generator dispatched 10000 packets per second to the SUT for 10 seconds. We disabled RSS in this measurement by not changing the source and destination addresses. At this level of pps, distributing CPU cores using RSS would worsen latencies and cause extra jitter.

The result is shown in Figure 6. Each data point on these graphs represents the average of 100,000 latency measurements. The latencies for `End.AN.NF`, `End`, and IPv4 are consistently 16.0 microseconds. In contrast, the combination of `End.DT4` and `H.Encaps` exhibits a latency of 19.0 microseconds. When measured in microsecond resolution, `End.AN.NF` matches the latency of `End` and IPv4 and is approximately 15.8% faster than the `End.DT4` and `H.Encaps` combination.

V. Conclusion

In this paper, we have proposed a new SRv6 End behavior, `End.AN.NF`, which integrates Linux netfilter and achieves the coexistence with routing protocols, such as BGP. `End.AN.NF` allows netfilter-based applications to serve as SR-Aware applications without modification, as `End.AN.NF` spoofs three netfilter hook points—prerouting, forward, and postrouting—to make them transparent to the SRv6 inner packet. Furthermore, `End.AN.NF` utilizes the `ARG` field in the SID to mark packets. This approach facilitates netfilter-based applications in matching marks on packet buffers, thereby allowing for dynamic rule adjustments. We implemented `End.AN.NF` on the Linux kernel and evaluated its performance. As a result, our implementation achieved 27% higher throughput and 3.0 microseconds lower latency than the combination of `End.DT4` and `H.Encaps`, which is one of the ways to apply netfilter rules to an SRv6 inner packet. Moreover, the difference in throughput between `End` and `End.AN.NF` was under 6%, indicating that the overhead of `End.AN.NF` is the acceptable range compared with the most basic End behavior.

References

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

[2] K. Kaur, V. Mangat, and K. Kumar, "A comprehensive survey of service function chain provisioning approaches in sdn and nfv architecture," *Computer Science Review*, vol. 38, p. 100298, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013720303981

[3] I. Trajkovska, M.-A. Kourtis, C. Sakkas, D. Baudinot, J. Silva, P. Harsh, G. Xylouris, T. M. Bohnert, and H. Koumaras, "Sdn-based service function chaining mechanism and service prototype implementation in nfv scenario," *Computer Standards & Interfaces*, vol. 54, pp. 247–265, 2017, sI: Standardization SDN&NFV. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S092054891730017X

[4] G. Davoli, W. Cerroni, C. Contoli, F. Foresta, and F. Callegati, "Implementation of service function chaining control plane through openflow," in *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2017, pp. 1–4.

[5] D. Bhamare, R. Jain, M. Samaka, and A. Erbad, "A survey on service function chaining," *Journal of Network and Computer Applications*, vol. 75, pp. 138–155, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804516301989

[6] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, p. 69–74, mar 2008. [Online]. Available: https://doi.org/10.1145/1355734.1355746

[7] P. Quinn, U. Elzur, and C. Pignataro, "Network Service Header (NSH)," RFC 8300, Jan. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8300

[8] A. Farrel, S. Bryant, and J. Drake, "An MPLS-Based Forwarding Plane for Service Function Chaining," RFC 8595, Jun. 2019. [Online]. Available: https://www.rfc-editor.org/info/rfc8595

[9] 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," Internet Engineering Task Force, Internet-Draft draft-ietf-spring-sr-service-programming-08, Aug. 2023, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-spring-sr-service-programming/08/

[10] R. Nakamura, Y. Ueno, and T. Kamata, "An Experiment of SRv6 Service Chaining at Interop Tokyo 2019 ShowNet," Internet Engineering Task Force, Internet-Draft draft-upa-srv6-service-chaining-exp-00, Oct. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-upa-srv6-service-chaining-exp/00/

[11] A. Abdelsalam, S. Salsano, F. Clad, P. Camarillo, and C. Filsfils, "Sera: Segment routing aware firewall for service function chaining scenarios," in *2018 IFIP Networking Conference (IFIP Networking) and Workshops*, 2018, pp. 46–54.

[12] C. Filsfils, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima, and Z. Li, "Segment Routing over IPv6 (SRv6) Network Programming," RFC 8986, Feb. 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc8986

[13] C. Filsfils, D. Dukes, S. Previdi, J. Leddy, S. Matsushima, and D. Voyer, "IPv6 Segment Routing Header (SRH)," RFC 8754, Mar. 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8754

[14] G. Dawra, K. Talaulikar, R. Raszuk, B. Decraene, S. Zhuang, and J. Rabadan, "BGP Overlay Services Based on Segment Routing over IPv6 (SRv6)," RFC 9252, Jul. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9252

[15] M. Haeberle, B. Steinert, M. Weiss, and M. Menth, "A caching sfc proxy based on ebpf," in *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, 2022, pp. 171–179.

[16] 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 *2019 IEEE Conference on Network Softwarization (NetSoft)*, 2019, pp. 333–341.

[17] B. Zhao, Y. Qin, W. Yang, P. Fan, and X. Zhou, "Sra: Leveraging af_xdp for programmable network functions with ipv6 segment routing," in *2022 IEEE 47th Conference on Local Computer Networks (LCN)*, 2022, pp. 455–462.

[18] The Netfilter's webmasters, "The netfilter.org "nftables" project." [Online]. Available: https://nftables.org/projects/nftables/index.html

[19] ——, "The netfilter.org "iptables" project." [Online]. Available: https://nftables.org/projects/iptables/index.html

[20] FRRouting Project, a Linux Foundation Collaborative Project, "Frrouting." [Online]. Available: https://frrouting.org/

[21] TRex Team, "Trex: Realistic traffic generator." [Online]. Available: https://trex-tgn.cisco.com/