GRAN SASSO SCIENCE INSTITUTE
SCHOOL OF ADVANCED STUDIES
Scuola Universitaria Superiore

PHD THESIS

Service Function Chaining with Segment Routing

PHD PROGRAM IN COMPUTER SCIENCE: XXXI CYCLE

PHD CANDIDATE
Ahmed Abdelsalam
Gran Sasso Science Institute

ADVISOR
Stefano Salsano
Università di Roma Tor Vergata

INTERNAL ADVISORS
Omar Inverso
Catia Trubiani
Gran Sasso Science Institute

August 2020

Gran Sasso Science Institute - GSSI

Abstract

Network Function Virtualization (NFV) is reshaping the way in which telecommunication networks and services are designed and operated. Traditional network functions are being transformed into Virtual Network Functions (VNFs) implemented as software modules. NFV promises a big capital expenditure (CAPEX) saving by replacing dedicated physical appliances with VNFs that can run on any commodity hardware. It also promises more flexibility as VNFs can be augmented with new features via simple software updates.

Service function chaining (SFC) denotes the process of forwarding packets through a sequence of network functions. It allows creating end-to-end services by composing several network functions. In traditional "pre-NFV" approaches, network functions were placed en-route, i.e. along the path of the flows. In NFV scenarios, VNFs can be arbitrarily located in the distributed virtualization infrastructure. Hence, SFC requires a steering method to force traffic to go through the chain of VNFs.

Segment Routing (SR) is a new network architecture based on source routing. It allows a node to steer packets through a set of network nodes in a specific order. In SR networks, a list of instructions, called *segments*, is attached to the packet to define such forwarding path. The SR architecture can be implemented with MPLS and IPv6 data plane. In this thesis, we only focus on the SR implementation based on IPv6 data plane, denoted as SRv6.

In this thesis, we leverage the SRv6 capabilities to design a scalable SRv6-based SFC architecture. We define the design principles of the architecture and discuss the challenges. We also propose the solutions to these challenges. Moreover, we study both the functional and non-functional properties of the architecture. More specifically, the contributions of this thesis are as follows:

- *Design of a scalable SFC architecture based on SRv6.*

  SFC is one of the most challenging use-cases in telecommunication networks. Traditional SFC solutions require maintaining per-chain state information in the network fabric to forward packets to VNFs. In the first part of Chapter 3, we propose a scalable SRv6-based SFC architecture. In the proposed architecture, each VNF is assigned a segment identifier (*SID*) and SFC is achieved by attaching a list of SIDs to the packets. Being SRv6-based, our approach does not need to maintain per-chain state information in the network fabric. Our solution is thus more scalable than the traditional alternatives. We define both data and control planes aspects of our proposed architecture. We also explain how to implement such architecture in a Linux environment.

- *Design and implementation of a solution to integrate legacy network functions into the SRv6-based SFC architecture.*

  In SRv6-based SFC, VNFs can be categorized into SRv6-aware and SRv6-unaware VNFs based on their ability to process SRv6 information in received packets. SRv6-unaware VNFs are legacy VNFs that can not process SRv6 packets. They might drop the packet or perform erroneous action. However, these SRv6-unaware VNFs have been in use since long time and network operators have spent a lot of efforts to automate their deployment and operations. In the second part of Chapter 3, we provide a solution to re-use such legacy SRv6-unaware VNFs within SRv6-based SFC architecture. To that end, we provide an SRv6-proxy that processes the SRv6 information on behalf of the SRv6-unaware VNFs. The proxy delivers plain IP packets to the VNFs with no SRv6 information. It restores the SRv6 information after the packets are processed by the VNF. The SRv6-proxy supports static, dynamic and masquerading behaviors to address the various types of VNFs. We provide an open source implementation for those proxy behaviors in Linux. We evaluate the processing overhead introduced by our SRv6-proxy implementation with respect to plain IP forwarding. The results are reported in Chapter 5 and show that our implementation provides a forwarding rate that can match the bandwidth requirements of VNFs.

- *Design and implementation of native SRv6-aware network functions.*

  With SRv6, the SR architecture has been evolving from the simple steering of packets across nodes to a general network programming approach. The idea is to encode instructions and not only locations in a segment list. In order to exploit such SRv6 "network programming" capabilities in SFC, network functions have to be SRv6-aware.

In Chapter 4, we design and implement several SRv6-aware network functions: SERA, SR-Snort, SR-nftables and SR-tcpdump. SERA is an SRv6-aware firewall capable of applying normal firewall behavior to packets with SRv6 information. It can also perform stateless SRv6-specific actions on packets. SERA is implemented as an extension to the commonly used iptables firewall. To the best of our knowledge, SERA is the first SRv6-aware network function ever realized. In SR-Snort, we extend the widely deployed open source IDS/IPS (Snort) to be SRv6-aware. SR-nftables, is another SRv6-aware network function which extends the next generation Linux firewall (nftables). SR-tcpdump is a tool that allows analyzing and debugging SRv6 traffic. It extends the tcpdump tool with a dissector for SRv6. The different SRv6-aware network functions developed for this thesis allow to build a fully automated SRv6-based SFC architecture. We contributed our implementations of SRv6-aware network functions to several open source projects to be used by network operators as well as other researchers. Several part of these implementations are currently integrated into the mainline of the Linux kernel, the iptables and nftables components and the tcpdump tool.

- *Design a performance evaluation framework for SRv6 implementations.*

  The SRv6 data plane implementations have been supported in many different routers implementations including: open source software routers such as the Linux kernel and the Vector Packet Processing (VPP) platform, as well as hardware implementations from different network vendors. Since then, SRv6 has been deployed both in service providers networks and data centers. It is therefore critical to assess some of the non-functional properties such as scalability and fault tolerance. In Chapter 5, we present SRPerf, a performance evaluation framework for SRv6 data plane implementations. The design of such framework is a very challenging task. As packets are required to be forwarded at an extremely high rate using a limited CPU budget to process each of them. We have used SRPerf to evaluate the performance of the SRv6 implementation in the Linux kernel and VPP. The framework allows us to identify some performance issues of the SRv6 implementation which we have fixed in new revisions.

Finally, we would like to highlight the tutorial on Linux kernel networking and SRv6 implementations provided in Appendix A. It could help other researchers to get started with these topics. In this respect, We also would like to mention our contribution to a survey and tutorial paper on Segment Routing that has been submitted.

Bibliographic notes

Conferences publications

1. 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"*, IEEE Conference on Network Softwarization (NetSoft), 2017.
2. A. Abdelsalam, S. Salsano, F. Clad, P. Camarillo, and C. Filsfils, *"SERA: SEgment Routing Aware Firewall for Service Function Chaining scenarios"*, IFIP Networking, 2018.
3. A. Abdelsalam, P. L. Ventre, A. Mayer, S. Salsano, P. Camarillo, F. Clad and C. Filsfils, *"Performance of IPv6 Segment Routing in Linux Kernel"*, 14th International Conference on Network and Service Management (CNSM), 2018.

Journal publications

1. A. Abdelsalam, P. L. Ventre, C. Scarpitta, S. Salsano, A. Mayer, P. Camarillo, F. Clad, C. Filsfils, *SRPerf: a Performance Evaluation Framework for IPv6 Segment Routing*, arXiv preprint, Submitted to IEEE Transactions on Network and Service Management (TNSM), 2020.
2. P. L. Ventre, S. Salsano, M. Polverini, A. Cianfrani, A. Abdelsalam, C. Filsfils, P. Camarillo, F. Clad, *Segment Routing: a Comprehensive Survey of Research Activities, Standardization Efforts and Implementation Results*, arXiv preprint, Submitted to IEEE Communications Surveys and Tutorials, 2019.

IETF contributions

1. C. Filsfils et al, *"SRv6 Network Programming"*, Internet Engineering Task Force (IETF), Internet-Draft draft-ietf-spring-srv6-network-programming, 2020.

2. F. Clad et al, *“Service Programming with Segment Routing”*, Internet Engineering Task Force (IETF), Internet-Draft draft-xuclad-spring-sr-service-programming, 2020.

3. C. Filsfils et al, *“SRv6 interoperability report”*, Internet Engineering Task Force (IETF), Internet-Draft draft-filsfils-spring-srv6-interop, 2019.

4. D. Voyer et al, *"Insertion of IPv6 Segment Routing Headers in a Controlled Domain"*, Internet Engineering Task Force (IETF), Internet-Draft "draft-voyer-6man-extension-header-insertion", 2020.

5. J. Guichard et al, *"Simplifying Firewall Rules with Network Programming and SRH Metadata"*, Internet Engineering Task Force (IETF), Internet-Draft "draft-guichard-spring-srv6-simplified-firewall", 2019.

6. K. Raza et al, "*YANG Data Model for SRv6 Base and Static"*, Internet Engineering Task Force (IETF), Internet-Draft "draft-raza-spring-srv6-yang", 2020.

Posters and demos

1. C. Filsfils, F. Clad, P. Camarillo, J. Liste, P. Jonnalagadda, M. Sharif, S. Salsano and A. AbdelSalam, *"Segment Routing IPv6 — Interoperability demo"*, SIGCOMM Industrial Demos, 2017.

2. A. Abdelsalam, *"Demo: Chaining of Segment Routing aware and unaware Service Functions"*, IFIP Networking, 2018.

3. A. Abdelsalam, S. Salsano, F. Clad, P. Camarillo, and C. Filsfils, *"SR-Snort: IPv6 Segment Routing Aware IDS/IPS"*, IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), 2018.

Talks

1. P. Camarillo and A. Abdelsalam, *"SRv6 Network Programming on FD.io VPP and Linux"*, Free and Open source Software Developers’ European Meeting (FOSDEM), 2018.

Acknowledgment

This thesis has been an incredible journey. During that journey I have met many people to whom I want to express my gratitude.

First and foremost, I would like to thank my advisor, Prof. Stefano Salsano. He has been continuously feeding me with new ideas, challenging me when it was time for review and acknowledging me when the work is done. This thesis would not exist without his advices and support.

I would like to thank my advisors from GSSI: Omar Inverso, Catia Trubinai. They have been always there to help. I'm very grateful to Omar for his time, help and advices.

I would like to thank Prof. Luca Aceto and Prof. Michele Flammini for their great support.

I would to thank my colleagues from University of Rome Tor Vergata: Paolo Lungaroni, Andrea Mayer, Carmine Scarpitta, Giuseppe Siracusano and Pier Luigi Ventre for all the work we achieved together during this thesis. It has been always nice to work with you.

I would like to thank colleagues from Cisco: Clarence Filsfils, Francois Clad, Pablo Camarillo and Jakub Horn for their support during my internship at Cisco and all the work we achieved together during this thesis. You are such an amazing team.

I would like to thank my friends from GSSI: Michele Aleandri, Ramiz Benyehia, Roberto Boccagna, Gennaro Ciampa, Lars Eric, Antonio Esposito, Wolfgang Haupt, Jon May, Sabir Ramazanov, Raffaele Scandone, Matteo Tonelli and Cosimo Vinci.

Last but not least, I would to like to thank my family. Deep in my heart, I love you.

Vi voglio veramente bene!

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Telecommunication networks connect billions of geographically distributed users and devices around the globe. In the last decades, they evolved from simple means to transfer either phone calls or data into converged networks that can carry any type of traffic such as voice and video. Moreover, in the last two decades we have seen a tremendous increase in the amount of traffic transmitted over these networks. The internet protocol (IP) has become the converged technology for these networks, therefore specialized IP networking devices, called *routers*, are required for the inter-connectivity between this vast number of users and forwarding such a big volume of traffic. Routers are able to connect multiple devices and networks. Each router has a routing table containing a set of route entries representing the shortest paths towards the networks of the same routing domain. Routing is the process of calculating such route entries, which can be either static or dynamic. Static route entries are manually configured by the network administrator, whereas dynamic routing entries are calculated by routing protocols.

A routing protocol is a distributed algorithm for finding the shortest path towards a given network. Routing protocols can be categorized into distance vector, link state or hybrid routing protocols. The distance vector protocols use the hop count (i.e., the number of intermediate points between the node doing routing and the destination of the packet) as metric for selecting one path over another. Link state protocols instead maintain a full map of the network topology and use an algorithm such as shortest path first (SPF) to select the best path towards a given destination. Hybrid routing protocols combine both techniques. The border gateway protocol (BGP) [1], open shortest path first (OSPF) [2], and intermediate system to intermediate system (IS-IS) [3] are currently the most commonly used routing protocols. Routing protocols are able to automatically discover new routes and calculate alternate paths in case of link failure, which makes them scale to large networks. However, one issue of dynamic routing is that applications have no way to express their preference on the routing path. Instead, one needs to implement a custom routing policy to force packets to go through a path other than shortest one calculated by dynamic routing protocols. Custom routing is useful in cases such as traffic engineering, load balancing, network programmability, fast re-route (*FRR*) [4], and service chaining [5]. The plain IP forwarding approach is not capable to support these scenarios, where custom routing is needed. Therefore IP has been combined with other networking technologies (e.g. ATM, MPLS) that are more suitable to support custom routing. Source routing is one way to implement such routing policy. It allows the source to define the packet path through the network. The source can be the packet originator or an ingress node of a network domain [5]. Source routing can be either strict or loose. In strict source routing, the source defines the full path that packets must follow through the network. In loose source routing, the source instead defines only some mid-points of the packet path [6]. In the IPv4 world, source routing is implemented as an IPv4 header option, named record route [7]. In the IPv6 world it is instead implemented as IPv6 routing extension header [8].

Routers, just as any other networking device, are usually built using a vertical tightly integrated model of both control and data (forwarding) planes. The control plane is the brain of the device that implements the logic of calculating the best path or any other special treatment for network traffic. The control plane programs the forwarding table of the data plane which has the job of forwarding the traffic at high speed according to these forwarding tables. However, this tightly integrated model of control and data planes does not allow network operator to innovate in their networks. Instead, for each feature, operators have to wait for the implementation from network devices vendors. This results in a vendor-lock, where operators are limited by the features delivered by the vendor. Software defined networking (SDN) is one way to re-think the way we build networking devices [9]. It eliminates the tight integration between the control and data planes, and thus allows to innovate independently either of them. SDN keeps the forwarding plane of a networking device very simple. The device needs to implement efficient data structures to store the forwarding information. The control plane is moved outside the devices into a central controller, called SDN controller, which has a full visibility of the whole network and can therefore calculate the forwarding paths. This allows to define a custom routing policies or adding new features by just updating the controller software. The SDN controller, after calculating the forwarding paths, programs the forwarding table of the devices. Devices need to provide an application programming interface (API) which the controller can remotely call to program the device. The API between the forwarding devices and the SDN controller is called southbound API. The *OpenFlow* protocol is an example of protocols used to program the forwarding devices [10].

The SDN technology has solved the problem of updating the forwarding devices with new features in a flexible and fast way. However, the big volume of traffic of converged networks requires networks to provide more services than just forwarding. These services include security, accounting, and quality of services to allow customized processing of the various types of traffic sent by different users with different traffic processing requirements. Middleboxes are networking devices designed to apply a special treatment to traffic other than forwarding. They can inspect, filter, or manipulate traffic in other ways. Firewalls, intrusion detection systems (IDS), network address translators (NAT), WAN optimizers, and load balancers are examples middleboxes which can be also referred to as *network functions*. Such network functions are usually deployed in the network as expensive proprietary hardware appliances. The proprietary nature of such hardware appliances increases the difficulty to introduce new network functions and services [11].

Network function virtualization (NFV) is dramatically changing the way in which networks are designed and operated [12]. Traditional specialized physical appliances are replaced with software module, known as virtual network functions (VNFs), that can run on any commodity hardware. A VNF can also be referred to as service function (SF). It can be deployed in a geographically distributed infrastructure. In NFV, the network function software is decoupled from the underlying hardware. Similar to SDN, the evolution of software and hardware is independent. This also provides a great level of flexibility in network function deployment, and makes it faster. The decoupling of hardware and software allows to scale network functions deployment. NFV promises a big capital expenditure (CAPEX) saving by replacing dedicated physical appliances with VNFs that can run on any commodity hardware.

The delivery of end-to-end services often requires the execution of various network functions. For example, a security service may include firewall, intrusion detection system (IDS), and deep packet inspection (DPI). Service function chaining (SFC) denotes the process of forwarding packets through a sequence of network functions [13]. There are several considerations to take when designing an SFC architecture, such as topological dependencies, configuration complexity, constrained high availability, consistent ordering of network functions, elastic service delivery, traffic selection criteria, classification and symmetric traffic flows [14]. One of the essential components of any SFC architecture is the steering mechanism that forces the traffic to go through the list of network functions. In traditional “pre-NFV” infrastructures, the physical appliances were placed en-route, i.e. along the path of the flows. This technique requires a significant effort for manual configurations, which makes it difficult to address the scaling and automation requirements of today's networks. In NFV scenarios, VNFs can be arbitrarily located in the distributed virtualization infrastructure. Hence, SFC requires a steering method to force traffic go through the chain of VNFs, which can be achieved using a source routing mechanism such as segment routing.

Segment Routing (SR) is a new network architecture based on source routing [15]. It allows a node to steer packets through a set of network nodes in a specific order. In SR networks, a list of instructions, called *segments*, are attached to the packet to define its forwarding path. The SR architecture can be implemented with MPLS and IPv6 data plane.

In this thesis, we leverage the SR implementation based on IPv6 data plane, denoted as SRv6, to tackle the SFC problem. In Chapter 2, we introduce the SR architecture including its data and control planes. In Chapter 3, we define an SFC architecture based on SRv6 and discuss how to integrate legacy network functions in this architecture. We introduce our design and implementation of native SRv6 network functions in Chapter 4. We evaluate the performance aspects of the various SRv6 implementations in Chapter 5. Finally, we report our conclusions in Chapter 6.

Chapter 2

Segment Routing

2.1 Introduction

Segment Routing (*SR*) is a new network architecture based on the source routing paradigm [15]. SR has shown a very successful standardization path within the internet engineering task force (IETF) [16]. The SR architecture and use-cases have been defined by the industry together with academic researchers [17]. Currently, the SR architecture can be instantiated on two different data planes: MPLS [18], also referred to as *SR-MPLS*, or IPv6 [8], which is referred to as *SRv6*. In this work, we focus on SRv6. SRv6 provides all the benefits of source routing in terms of flexibility and simplicity needed for today's networks, but also addresses the security concerns of previously defined source routing techniques for IPv6 [19] [20]. The collaboration between research and industry resulted in the support of SRv6 implementations across several platforms such as the Linux kernel, VPP [21], and hardware from various network vendors [22], which are fully inter-operable as reported in [23].

In this chapter, we introduce the SR technology. In Section 2.2, we discuss the data and control planes of the SR architecture (both for SR-MPLS and SRv6). From Section 2.3, we focus on SRv6, covering the segment routing header (SRH) which is the main building block of SR in IPv6 networks. Finally, we explore the SRv6 network programming model in Section 2.4.

2.2 Segment Routing architecture

The SR architecture relies on the source routing paradigm. The source can attach to a packet an ordered list of instructions, denoted as *segments*, to steer the packet through a set of intermediate steps in the path towards its final destination. In SR networks, the list of segments identifiers (SIDs) are attached to packets as a part of an extra header, which carries also a pointer to the next segment to be processed. SIDs may represent a topological or service instructions. Topological SIDs might have local or global significance. Locally significant topological SIDs define the forwarding behavior of a node. Globally significant topological SIDs define a forwarding path to a destination through an SR domain and they must be unique.

There are three major variants of a global SID: *prefix* SID which is associated with a prefix, *node* SID which is allocated to a loopback that identifies a specific node, and *anycast* SID which is assigned to a loopback shared by a set of routers. It is possible to have more than one topological SID for the same destination, each of them defining a different path with a different objective (e.g., low-latency path or high-throughput path). On the other hand, service SIDs define the services (e.g., container or virtual machine) that should process the packet. In the next subsections, we consider the two different SR data planes: *SR-MPLS* based on MPLS [18], and *SRv6* based on IPv6 [8].

2.2.1 SR-MPLS

SR-MPLS is the SR architecture instantiation over the MPLS data plane. It requires no change in the MPLS forwarding behavior [24]. SIDs are represented as MPLS labels and the SIDs list is represented as a stack of MPLS labels. The data plane on an SR-MPLS nodes supports three main operations: *push* consists in pushing one or more SIDs on the top of a received packet before forwarding it to the next-hop in the path towards its destination, *continue* swaps the incoming SID with an outgoing SID, and *next* pops the topmost SID [25].

An example of traffic engineering (TE) use-case is shown in Figure 2.1. The SR-MPLS is domain is composed of four MPLS routers (N1, N2, N3 and N4). Each router is assigned an MPLS label in the form 910N, where N represents the router number. For example, router $N2$ is assigned the MPLS label 9102. The SR-MPLS network offers several paths to connect hosts A and B, each with different latency and throughput characteristics. For example, the path $N1 \rightarrow N2 \rightarrow N3 \rightarrow N4$ provides very high throughput as shown in Figure 2.1. However, this path is not the shortest path between between A and B. In order to enforce the traffic to go through such path, router $N1$ is programmed with a policy that pushes a list of SR SIDs representing the nodes of that path. Once this SID list is added to packets, all the SR-MPLS routers do not need to keep any state regarding the traffic, as the state is carried in the packet. This provides a great simplification in the network, as the state is maintained only at the edge routers and added to the packets. Nodes in the SR-MPLS network are only required to remove the top SID and send the packet to the next hop in the SID list.

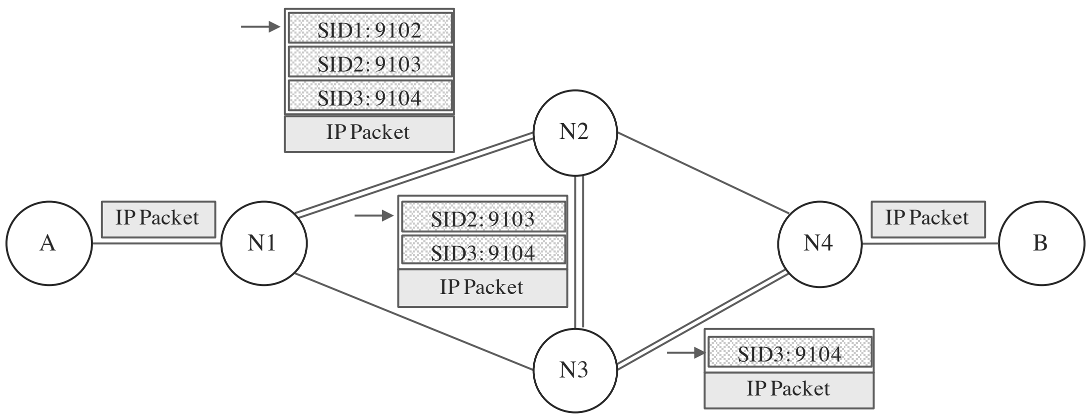


Figure 2.1: Traffic engineering with SR-MPLS

2.2.2 SRv6

SRv6 is the SR architecture instantiation over the IPv6 data plane. It defines a new type of IPv6 routing extension header, named segment routing header (SRH), that is used to carry the SID list. Each SID is represented as an IPv6 address. IPv6 addresses at a given node are not SRv6 SIDs by default, but they need to be explicitly instantiated as SRv6 SIDs. In SRv6 domain, not all nodes are required to support SRv6, since the IPv6 routing extension header is not examined or processed by any node along a packet's delivery path until the packet reaches the node identified in the destination address field of the IPv6 header [8]. Accordingly, only nodes represented with SIDs in the SID list are required to support SRv6. Packets follow the shortest path computed by the interior gateway protocol (IGP) between nodes represented in the SID list.

2.2.3 SR control plane

The SR control plane is concerned with communicating SID information among devices in the network, and defining the list of SIDs to be added to the packet at the source. The SR architecture supports any type of control plane: distributed, centralized, or hybrid. The SIDs can be allocated and signaled by the routing protocol (e.g., OSPF or BGP). The SR controller is responsible for the allocation of SIDs as well as computing the SR policies. It listens to all link updates from all routers in the SR domain as shown in Figure 2.2. The SR controller has a full view of the network and is responsible for calculating the SID list for a given traffic requirements. The ingress (source) node communicates with the SR controller to get the SID list for a given traffic flow. Then, it steers through that SID list. The SR controller can program the devices in the SR network using any of the available southbound interfaces (e.g., NETCONF [26], PCEP [27], BGP [1]). Open-Daylight is an example of available SR controllers which uses PCEP as a southbound interface [28].

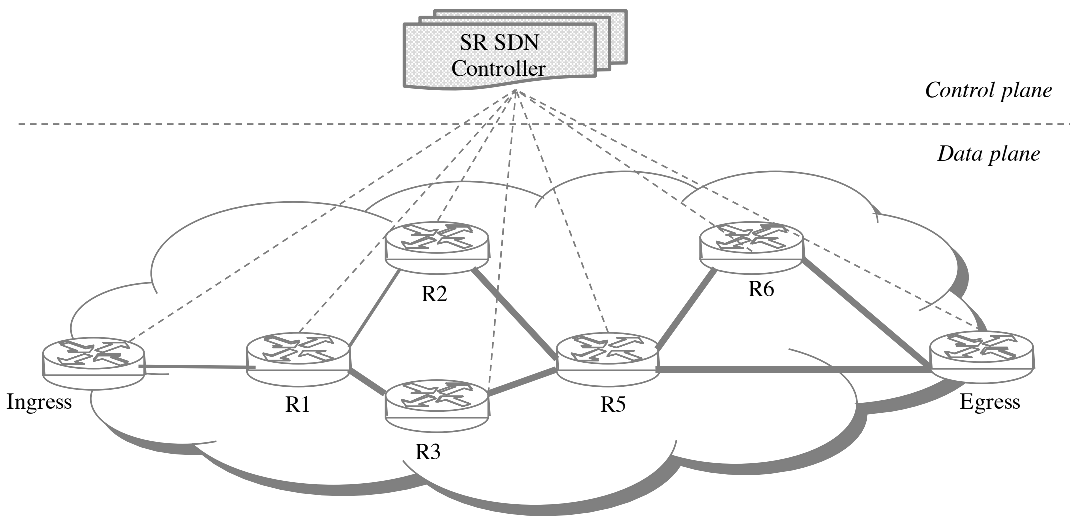


Figure 2.2: SR Control Plane

2.3 Segment Routing Header

IPv6 extension headers are used to encode optional internet-layer information. They are placed between the IPv6 header and the upper layer header (e.g., TCP or UDP) of a packet as highlighted by double lines in Figure 2.3. IPv6 extension headers are identified by a *Next Header* value assigned by the IANA [29].

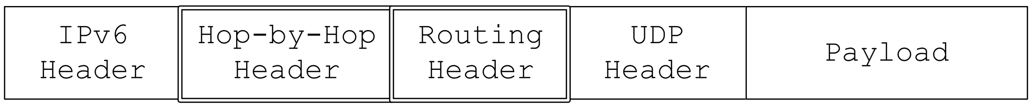


Figure 2.3: IPv6 packet with extension headers

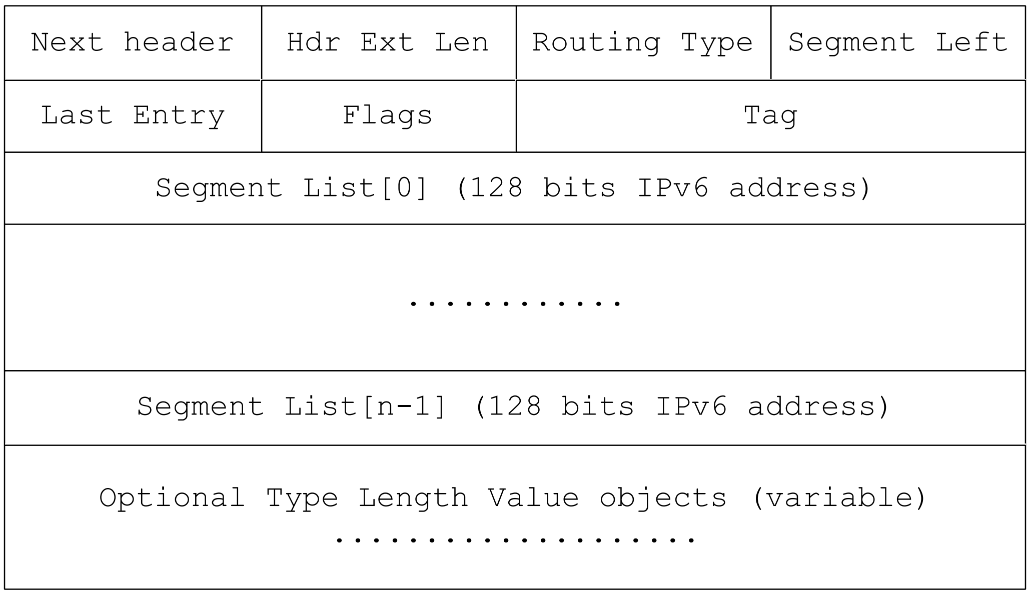

Next header
Hdr Ext Len
Routing Type
Segment Left
Last Entry
Flags
Tag
Segment List[0] (128 bits IPv6 address)
....
Segment List[n-1] (128 bits IPv6 address)
Optional Type Length Value objects (variable)
....

Figure 2.4: IPv6 segment routing header (SRH)

Nodes supporting IPv6 should parse extension headers until the upper layer protocol is found (i.e., the first *Next Header* value that does not correspond to an extension header). The currently defined IPv6 extension headers are *Hop-by-Hop options*, *Fragment*, *Destination Options*, *Routing*, *Authentication*, and *Encapsulating Security Payload*. IPv6 routing extension headers are used in IPv6 networks to implement source routing. They are assigned 43 as a protocol number value. Several types of routing extension headers are defined to support several use-cases including type-0 that was deprecated for security issues [20] and type-2 that supports IPv6 mobility [30].

SRH is a new type of the IPv6 routing extension header with type value 4. The SRH format is defined in [31]. Figure 2.4 shows the format of the SRH and its fields which can be described as follows:

- Next Header: 8-bit field identifying the type of header immediately following the SRH. The Next Header value is assigned based on the IANA [32]
- Hdr Ext Len: 8-bit field representing the length of SRH in 8-octet units (not including the first 8 octets).
- Routing Type: 8-bit field identifying the routing header type assigned for SRH.
- Segments Left: 8-bit field identifying the number of remaining SIDs that still need to be traversed by the packet before being delivered to its final destination.
- Last Entry: 8-bit field carrying the index of the last SID in the SID list, which is the first SID to be traversed since the SID list is traversed in the reverse order.
- Flags: 8-bit field carrying some flags. Each flag is represented by one bit and may triggers a special processing for the SR packet. The presence of a flag is represented by a value of 1 in the corresponding position.
- Tag: 16-bit field to identify a packet as part of a class or group of packets. When tag is set to zero, it means that it is not used and should be ignored by the receiving node.
- Segment List[n]: 128-bit IPv6 address representing the n-th segment in the SIDs list. The SIDs list is encoded starting from the last SID of the SR policy. The first element of the SIDs list contains the last SID of the SR policy. The second element contains the penultimate SID of the SR policy, and so on.
- Type Length Value (TLV): variable length field used to carry some optional information along with the SRH as explained in Section 2.3.1.

2.3.1 SRH TLVs

Type length value (TLV) is a common format to encode optional information of a given protocol [33]. The type and length fields are fixed in size and the value field is of variable size. The type field identifies the type of TLV, the length field identifies the size of the succeeding value field, and the value field carries the information for the encoded option. SRH supports a set of TLVs that can be used to encode some optional information. A SRH can carry one or more TLVs for the same packet. Nodes processing the TLV should simply ignore any TLV that has an unrecognized type. Figure 2.5 represents the generic SRH TLVs format where:

- Type: 8-bit field identifying the type of TLV.
- Length: 8-bit field identifying the length of the TLV's variable length data.
- Variable length data: actual data used to encode optional information.

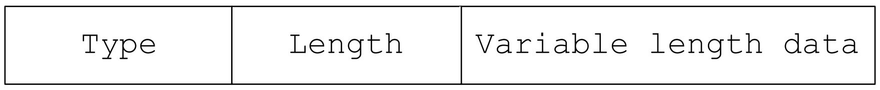


Figure 2.5: Generic format of SRH TLVs

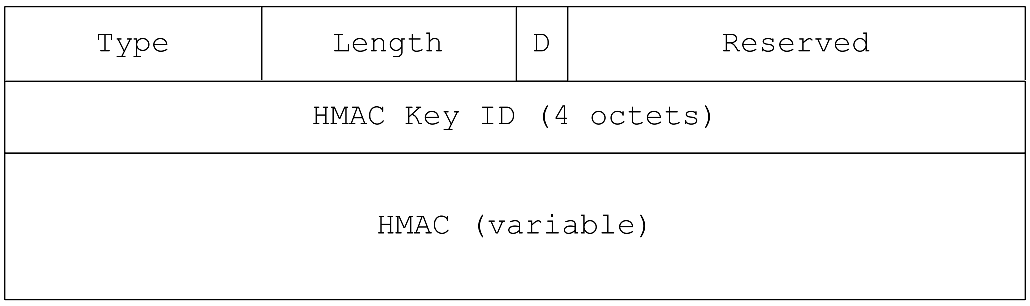


Figure 2.6: HMAC TLV format

2.3.2 HMAC TLV

The SRH defines HMAC TLV to address the security issues related to source routing. It is used to verify the validation and integrity of an SR packet. The HMAC TLV format is shown in Figure 2.6 where:

- Type: 8-bit field identifying the HMAC TLV type (suggested value 5).
- Length: 8-bit field representing the HMAC TLV length in bytes (38).
- D: 1 bit, set to 1 to indicate that the destination address verification is disabled due to use of reduced segment list.
- RESERVED: 15 bits. MUST be 0 on transmission and ignored on receipt.

- HMAC Key ID: A 4 octet opaque number which uniquely identifies the pre-shared key and algorithm used to generate the HMAC.
- HMAC: keyed HMAC, in multiples of 8 octets, at most 32 octets.

Nodes outside the SR domain are not able to compute the right HMAC since the computation requires a pre-shared key. Packets with wrong HMAC value are rejected at the SR ingress router. Adding an HMAC to each and every SR packet increases the security, but it has a performance impact. However, only the SR ingress node needs to process the HMAC TLV and all other SR nodes can ignore the HMAC TLV since they trust the SR ingress router. This speeds up the forwarding because SR routers which do not validate the SRH do not need to parse the SRH until the end. More detailed security considerations related to SRH can be found in the specific section of [31]

2.3.3 SRH processing

The SRH processing in an SR domain is based on the type of the SR node which can be: SR source node, SR transit node, or SR endpoint node. The SR source node is the node that originates SRv6 packets, it can be either the host originating the IPv6 packet or the ingress node of an SR domain. It can be configured with one or more SR policies. Packets are steered through one of the configured SR policies. The first SID of the SID list associated with an SR policy is encoded as the packet destination address. Then the packet is forwarded towards its destination address. The SR transit node is a node that forwards SRv6 traffic where the destination address of the packet is not configured as local SID. It can be SR-capable or SR-incapable. SR transit nodes are not required to inspect the SRH of received SR packets since the destination address of the packet does not correspond to locally configured SID or interface. They forward packet towards the next hop in the path towards the destination. The SR endpoint node is a node that receives an SRv6 packet where the destination address of that packet is locally configured as a local SID. It must process the SRH and the processing is based on the *Segments Left* value. If *Segments Left* is equal to zero, then it proceeds to process the next header in the packet. In case of non zero value of Segments Left, the *Segments Left* is decremented by 1. Then the packet's destination is address is updated to be the active SID identified by *Segments Left*. Finally Hop Limit is decremented and the packet is handed again to the routing module to be forwarded based on the new destination.

2.4 SRv6 network programming

The SRv6 architecture has been extended from the simple steering of packets across nodes to a general network programming model [34]. The idea is to encode in the SID list not just locations, but also the processing behavior to be executed by the nodes receiving the packet. This is feasible thanks to the huge IPv6 addressing space. Each SR node implements a table containing the SIDs local to that node. Such table is known as *local SID table*, which might be a separate table or part of the main forwarding table. Each entry of the *local SID table* identifies the function associated with the local SID and its parameters. The SRv6 network programming model defines a set of processing behaviors that can be associated with an SRv6 SID. The set of SRv6 behaviors can be categorized into *endpoint* and *transit* behaviors.

2.4.1 Transit behaviors

SR source nodes, where SRv6 encapsulated packets are originated, are often configured with one or more SRv6 policies. Each SRv6 policy includes a SID list identifying a specific path through the network. SRv6 transit behaviors steer received packets (or layer-2 frames) into an SRv6 policy. When SR source nodes receive packets that match a configured SRv6 policy, the transit behavior bound to the policy is applied to those packets. The currently supported SRv6 transit behaviors (modes) in the networking programming model are shown in Figure 2.7 and defined as follows:

- *T.Insert* is the transit behavior with insertion of an SRv6 policy. In this mode the SRH is inserted in the original IPv6 packet, immediately after the IPv6 header and before the transport level header. The original IPv6 header is modified, in particular the IPv6 destination address is replaced with the IPv6 address of the first segment in the segment list, while the original IPv6 destination address is carried in the SRH header as the last segment of the segment list.

- *T.Encaps*: the transit behavior with encapsulation in an SRv6 policy. In the encap mode the original IPv6 packet is carried as the inner packet of an IPv6-in-IPv6 encapsulated packet. The outer IPv6 packet carries the SRH header with the segment list.

- *T.Encaps.L2*: the transit behavior with encapsulation of layer-2 frame in an SRv6 policy. It works the same as the *T.Encaps* behavior, with the difference that *T.Encaps.L2* encapsulates the whole received layer-2 frame rather than the packet.

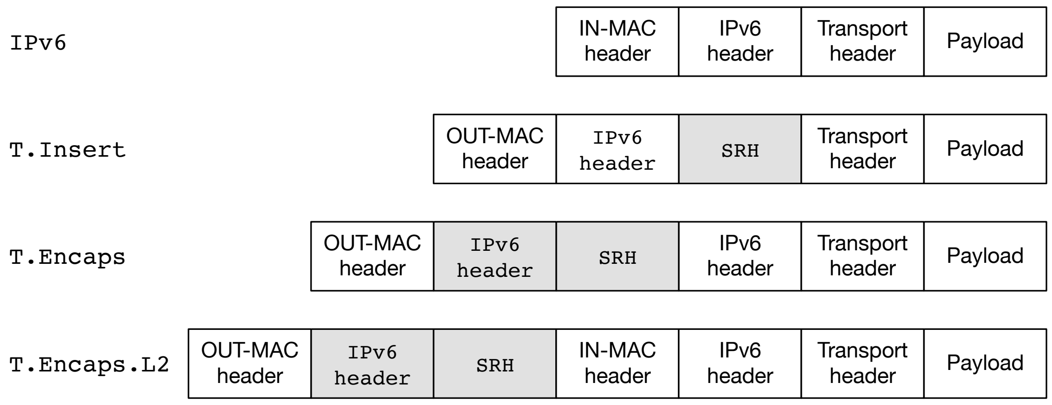


Figure 2.7: SRv6 Transit Behaviors

2.4.2 Endpoint behaviors

For each SRv6 SID configured in a SR endpoint node, there is a function to be executed on packets matching the SID. In [34], a set of well-known functions that can be associated with a segment are defined including:

- *End* is the endpoint function. This is the most basic function. Firstly, the Segments Left of the SRH is decreased. The IPv6 destination address of the packet is replaced with the next active segment. Then, a FIB lookup using the updated on the updated destination address. Finally, the packet is forwarded according to the matched FIB entry. This behavior represents the SRv6 instantiation of a prefix SID.

- *End.X* is the endpoint with layer-3 cross-connect to an IPv6 adjacency, which represents the SRv6 instantiation of an adjacency SID. It is a variant of the *End* function where the packet is forwarded to layer-3 adjacency bound to the SID rather than the IPv6 destination address.

- *End.T* is the endpoint with specific IPv6 table lookup. It is a variant of the *End* function where the FIB lookup is performed in an IPv6 table associated with the SID rather than the main FIB table. The *End.T* is used for multi-table operation in the network core.

- *End.DX2* is the endpoint with decapsulation and layer-2 cross-connect to an output interface. It is used to implement the L2VPN use-cases. It is a variant of the endpoint function where the outer IPv6 header and its extension headers are popped and the resulting frame is forwarded via the output interface associated to the SID.

- *End.DX4* is the endpoint with decapsulation and IPv4 cross-connect to IPv4 adjacency. It is used to implement the L3VPN use-cases where a FIB lookup in a specific tenant table at the egress node is not required. It is a variant of the *End* and *End.X* functions where the outer IPv6 header and its extension headers are popped and the resulting IPv4 packet is forwarded to layer-3 adjacency bound to the SID.

- *End.DX6* is the endpoint with decapsulation and IPv6 cross-connect to an IPv6 adjacency. It is used to implement the L3VPN use-cases where a FIB lookup in a specific tenant table at the egress node is not required. It is a variant of the *End* and *End.X* functions where the outer IPv6 header and its extension headers are popped and the resulting IPv6 packet is forwarded to layer-3 adjacency bound to the SID.

- *End.DT6* is the endpoint with decapsulation and specific IPv6 table. It is used to implement the L3VPN use-cases where a FIB lookup in a specific tenant table at the egress node is required. It is a variant of the *End* function where the outer IPv6 header and its extension headers are popped and lookup for the exposed inner IPv6 destination address is performed in an IPv6 table associated with the SID.

2.4.3 SRv6 support in Linux

The SRv6 capabilities were firstly merged in Linux kernel 4.10 [35]. The implementation supported the SRv6 *End* behavior. The SRv6 *T.Encaps* and *T.insert* behaviors were also supported. Kernel 4.14 is another milestone for SRv6 support in Linux kernel. A new set of SRv6 behaviors has been added to the kernel [36]. The supported SRv6 Endpoint behaviors included in kernel 4.14 are: *End.X*, *End.T*, *End.DX2*, *End.DX4*, *End.DX6*, *End.DT6*, *End.B6*, *End.B6.Encaps*. Along with the newly supported SRv6 Endpoint behaviors, some new transit behaviors have been added including: *T.Encaps4* which steers IPv4 packets through an SRv6 policy and *T.Encaps.L2* which encapsulates a packet along with the L2 frame (i.e. the received Ethernet header and its optional VLAN header) in the payload of the outer IPv6 packet.

The implementation details of the SRv6 behaviors are explained in Appendix A and their performance in reported in Chapter 5.

Chapter 3

Service Function Chaining

3.1 Introduction

Service Function Chaining (SFC) denotes the process of forwarding packets through a sequence of service functions [13]. The SFC process is not straight-forward: many aspects need to be considered, such as topological dependencies, configuration complexity, constrained high availability, consistent ordering of service functions, traffic selection criteria, (re)classification per service function, limited end-to-end service visibility, symmetric traffic flows, and multi-vendor service functions (SFs).

The IETF SFC working group has defined an SFC reference architecture [13]. The architecture includes the concepts, principles, and components required to deploy SFC. An essential component of an SFC deployment is the steering mechanism that forces packets to go through the SFs that encompass the SFC. The steering is often achieved by attaching an extra header with topological information to be applied to the packets. Such process is referred to as SFC encapsulation and the resulting packet are called SFC-encapsulated packet.

In this chapter, we provide a solution for SFC based on the SRv6 network programming model. This chapter is based on previously published work [37] [38] [39] [40] [41]. This chapter is structured as follows, we define Service Function Chaining (SFC) in Section 3.2. Then, we describe the architectural concepts, principles, and components of the SFC architecture in Section 3.3. We explain the proposed SRv6-based SFC architecture is explained in Section 3.4. In Section 3.5, we elaborate on the difference between SRv6-aware and SRv6-unware service functions, and propose our solution for chaining SRv6-unaware functions. We present the implementation of the proposed architecture in Section 3.6. Finally, we conclude the chapter with the state of the art of SFC in Section 3.7 .

3.2 Service Function Chaining

Service Function Chaining (SFC) allows the delivery of advanced end-to-end services composed of one or more service functions. The service functions can be traditional network service functions, such as firewalls or load balancers, as well as application-specific features such as HTTP header manipulation. The current SFC deployment models are tightly coupled to the network topology and to the physical resources [14]. This leads to static deployments and thus hinders the NFV business goals of greater business agility, fast time-to-market, improved business processes, optimized operating expenses (OPEX), and lower capital expenditures (CAPEX) [42].

Several aspects have to be addressed to achieve successful SFC deployment. These aspects are described in [14], which can be summarized as follows:

- **Topological dependencies**. The deployment of network services is often coupled to the network topology, whether it be physical, virtualized, or a hybrid of the two. Such topological coupling limits both the placement and selection of service functions. Therefore, placement and service function selection that take into account network topology information such as, traffic load or traffic engineering are becoming hardly feasible. In addition, such dependency imposes constraints on service delivery, and reduces flexibility. This affects scale, capacity, and redundancy across network resources.
- **Configuration complexity.** The configurations required to deploy SFCs can be quite complex as a consequence of the topological dependencies. For each simple change in the deployment of a given SFC, there may be changes to the logical and/or physical topology. Accordingly, service delivery deployments are becoming very static and slow, since network operators may not be willing to take the risk of making changes in their topology.
- **Constrained high availability.** With static deployment, traffic reaches service functions based on the network topology. Accordingly, redundant service functions must be placed in the same topology as the primary service. This limits the availability of service functions as a consequence of the topological dependencies.

- **Consistent ordering of service functions.** Service functions are typically independent. However, for SFC deployments the order in which service function should be executed is very important. Specially in cases of topological dependent deployment in order to wire the service functions of a given SFC together. Currently, there is no consistent way to enforce specific ordering of the service functions within a given SFC.

- **Transport dependence.** Service functions are deployed in networks with a range of different network underlays and overlays, such as Generic Routing Encapsulation (GRE) [43], Virtual eXtensible Local Area Network (VXLAN) [44], MPLS [18], and SRv6 [34]. Hence, service functions are required to support many transport encapsulations to address the issue of coupling of service functions to the topology.

- **Elastic service delivery.** Realizing the rapid changes of service deployment in terms of capacity can be a very challenging task, since it relies mostly on routing modification, which is a risky and complex process for network operators.

- **Traffic selection criteria.** Selecting a part of the traffic to be processed by a given service function(s) is not a trivial process. One of the common techniques to address this problem is to send all traffic of a given network segment to be processed by a pre-defined set of service functions. This might lead to a situation where service functions are overwhelmed by traffic that does not need to be processed. An alternative solution is to use Policy Based Routing (PBR) [45] to achieve more granular traffic control. However, such PBR techniques are require significant amount of complex configurations.

- **Limited end-to-end visibility.** Service function chains can span multiple data centers, which makes troubleshooting very complex due to the limited end-to-end service visibility. Furthermore, the separation between physical and virtual environments can add further complexity to the overall process.

- **(Re)classification per service function.** Due to the lack of efficient mechanisms to share the details of classification information between services, the classification process occurs at each service function without leveraging the previously applied service functions.

- **Symmetric traffic flows.** Service function chains can be unidirectional, where traffic is passed through a set of service functions in one forwarding direction only, or bidirectional, where traffic flows through a set of service functions in both forwarding directions. Many common service functions such as DPI and firewalls often require bidirectional chaining in order to ensure that the flow state is consistent. Realizing such bidirectional service function chains requires very complex configuration due to the static nature of current deployment models.

- **Multi-vendor service functions.** Network operators often rely on service functions from different network vendors to deploy SFC. This requires per vendor-specific expertise, and calls for standards to ensure interoperability between services from different vendors.

3.3 SFC architecture

RFC 7665 defines a high-level SFC architecture to address the aspects described in the SFC problem statement [13]. In this architecture each service function is treated as an opaque processing element. Generally, the list of enabled service functions in a given domain is not pre-defined and may vary with time. The set of service functions required to deploy an SFC can vary from one administrative entity to another. In addition, each administrative domain may have several SFC policies that can be simultaneously applied to meet various business requirements.

The SFC architecture puts no constraints on the underlay technology to be used, but it is up to the network operator to decide the underlay technology to realize the SFC architecture. Service functions (SFs) are referenced using a unique identifier within the SF domain and the ordered set of SFs to be applied to packets are determined based on a classification process. SFC provides more than just the application of an ordered set of SFs to selected traffic; rather, it describes a method for deploying SFs in a way that enables dynamic ordering and topological independence of those SFs as well as the exchange of metadata between participating entities.

SFCs can be unidirectional, bidirectional, or a hybrid of both. They may contain cycles, where some SFs process traffic several times. SFC uses the *Service Function Path (SFP)* as a mechanism to express the forwarding path of traffic for a given chain. A single SFC can have multiple associated SFPs. The SFC architecture does not mandate the degree of granularity of the SFP. The SFP granularity depends on the specific SFC solution being used. SFPs can thus be quite vague, but in some cases they can be fully specified by defining all SFs as well as the intermediate nodes where the traffic should go between the different SFs.

The architecture defines some key architectural principles; topological independence, plane separation, classification, shared metadata, service definition independence and SFC independence. SFC deployment should not require any changes in the underlay network. The SFP definition should be separate from the actual packet forwarding. Traffic should be forwarded through a specific SFP, if it matches the set of classification rules associated with that SPF. Classification can occur at multiple points within an SFC, and it can have varying degrees of granularity such 5-tuple of a packet. Metadata can be shared among SFs and classifiers and between external systems and SFs. Such metadata provides a mean to communicate the results of preceding processing to the subsequent ones. The SFC architecture has to be independent from the actual implementation details of the SFs themselves. Finally, the creation, modification, or deletion of an SFC should have no impact on other SFCs.

In SFC, classifiers are configured with different policies that meet the different traffic requirement. Each policy includes a set of classification criteria. Classifiers compare received traffic to the policies, and traffic that satisfies the classification criteria of a policy is directed into the SFP associated with that policy. Classification initially occurs at the ingress of the SFC domain. The granularity of the initial classification is determined by the capabilities of the classifier and the requirements of the SFC policy. As a consequence of the classification, the appropriate SFC encapsulation is imposed on the data, and a suitable SFP is selected or created. The SFC architecture supports reclassification as well. As packets traverse an SFP, reclassification may occur. Reclassification may result in the selection of a new SFP, an update of the associated metadata, or both. This is referred to as "branching".

SFs are the core element of SFC and can be categorized into SFC-aware and SFC-unaware based on their ability to process the SFC encapsulation. SFC-aware SFs can process SFC-encapsulated packets, meaning that they are able to act on the original packet. On the contrary, SFC-unaware SFs are not able to process SFC-encapsulated packets. SFC proxies allow to include SFC-unaware SFs into the SFC architecture. The SFC proxy acts as a gateway between the SFC encapsulation and SFC-unaware SFs. The proxy accepts packets on behalf of the SF. It removes the SFC encapsulation and delivers IP packet to SFC-unaware SFs. For the packets coming out from an SFC-unaware SF, the proxy can receive them, re-apply the SFC encapsulation, and return them to the service function forwarder (SFF) for processing along the service function path. Otherwise the packets could be reclassified from scratch and associated to the proper SFC.

3.4 SRv6 based SFC Architecture

3.4.1 Design principles

The IETF SFC working group introduces the generic concept of SFC encapsulation without specifying the protocols required to enforce the service forwarding path (SFP). In this thesis, we propose the use of IPv6 segment routing (SRv6) to support SFC. SRv6 allows to steer packets through an ordered list of instructions, called segments. These segments may encode simple routing instructions for forwarding packets through a specific network path, or rich behaviors to support use-cases such as service chaining.

In the context of service chaining, each service, running either on a physical appliance or in a virtual environment, is associated with a segment, which can be included in a segment list to steer packets through the service. Such service segments may be combined together in a segment list to achieve service chaining, but also with other types of segments as defined in [15]. SRv6 thus provides a fully integrated solution for service chaining, overlay and underlay optimization. Furthermore, the IPv6 data plane natively supports metadata transportation as part of the SRv6 information attached to the packets.

In the rest of this section, we will refer to the Service Functions (SF) that we have introduced earlier as *Virtual Network Functions (VNFs).* This is the terminology used in the context of *Network Function Virtualization (NFV).* As discussed in general for SFs, VNFs will be divided into two classes depending on whether they are able to behave properly in the presence of SRv6 information. These are respectively named SRv6-aware and SRv6-unaware VNFs. An SRv6-aware VNF can process the SRv6 information from the packets it receives. This means that is able to identify the active segment as a local instruction and move forward in the segment list, but also that the VNFs own behavior is not hindered by the presence of SRv6 information. SRv6-aware VNFs can process the information contained in the SRH of incoming packets and can use the SRH to influence the processing/forwarding of the outgoing packets. Any VNF that does not meet these criteria is considered as SRv6-unaware. SRv6-unaware VNFs are not capable to understand the SRH, they can only reason in terms of traditional IP operations.

A typical case of SRv6-unaware VNFs is the case in which a pre-existing VNF (also referred to as a *legacy* VNF) is used in an SRv6-based SFC scenario. In this case, legacy VNFs need to be inserted in the SFC processing chain in such a way that they can receive, process and forward plain IP packets with no knowledge of the SRH and of the SFC infrastructure. In this case, the SFC infrastructure needs to take care of handing the packets to the SRv6-unaware

VNFs and to receive the packets from them, performing the adaptation with the SRv6-based SFC processing chain. The SRv6-based SFC architecture is divided into data plane and control plane components as depicted in Figure 3.1.

3.4.2 Data plane

In the data plane, the network is composed of IPv6 routers and NFV nodes as shown in Figure 3.1. IPv6 routers can be SRv6-enabled or legacy IPv6 router. SRv6-enabled routers (*Ingress* and *Egress*) are able to add, remove, or process the SRv6 encapsulation. Legacy IPv6 routers (*R1* and *R2*) simply forward packets based on their destination address regardless of the presence of the SRH [1]. NFV nodes (*NFV1* and *NFV2*) can host some VNFs running as VMs and/or containers. NFV nodes can be SRv6-enabled or legacy IPv6 routers. Each VNF instance running in a NFV node is uniquely identified by an IPv6 address.

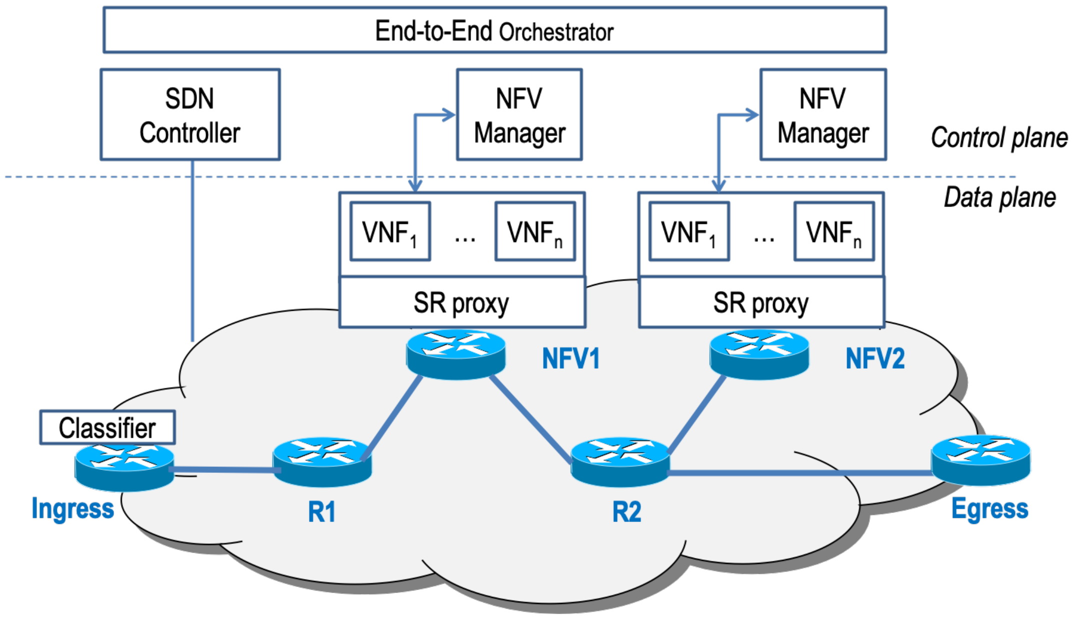


Figure 3.1: SRv6 SFC Architecture.

Typically, VNFs running in the same NFV node will be assigned IPv6 addresses from the same IPv6 prefix. Therefore, traffic destined to those VNFs is routed to their hosting NFV node using the prefix of that node. For example, VNFs 1 through k running in NFV node N will be assigned an IPv6 address in the form $N :: F_1/64$ through $N :: F_k/64$ and traffic destined to

[1]Extension headers, except hop-by-hop, are not examined or processed by any node along a packet's delivery path, until the packet reaches the node identified in the destination address field of the IPv6 header [8].

those VNFs is typically routed using the prefix $N :: /64$. We represent an SFC as <v_1,v_2,...,v_n>, where v_i is the IPv6 address of the i-th VNF in the chain. Each VNF IPv6 address corresponds to an SRv6 SID and the SFC is represented by an ordered list of segments encoded in the IPv6 SRH.

An SRv6-based SFC domain is bounded by one or more SRv6 edge routers. There are two types of SRv6 edge routers, known as *Ingress* and *Egress*. An SRv6 *Ingress* router classifies the incoming packets and associates them to a VNF chain. Then the packets are steered through the SFP associated with that VNF chain, which implies attaching a list of SRv6 SIDs to be traversed. The attaching of SRv6 SID list to the packets can be done into two modes: *Encap* or *Insert* [31]. In the *Encap* mode the original IPv6 packet is used as payload of a new packet composed of a outer IPv6 header, the SRH, and the original packet. The outer IPv6 header has the ingress router IPv6 address as source address, the next VNF in the chain as IPv6 destination address, and the egress router as last segment in the segment list. In the *Insert* mode the SRH is added directly to the original packet. The IPv6 destination address is set to the next VNF in the chain and the original destination address of the packet is added as the last SID in the segments list.

In case of the Encap mode, SRv6 *Egress* routers are responsible for removing the SRv6 encapsulation from packets before they leave the SRv6 domain. SRv6 encapsulation is removed because it carries topological information of the SRv6 domain that should not be exposed to the outside. At the NFV node, when a packet arrives with the destination address equals to one of the VNFs running inside the node, the packet processing will be based on type of the VNF, i.e., SRv6-aware or SRv6-unaware. For SRv6-aware VNFs the packet is delivered unchanged. In case of SRv6-unaware VNFs, the SRv6 proxy will handle the processing of the SR information on behalf of the SRv6-unaware VNFs. After the packet has been successfully processed by all VNFs of a given chain, it is forwarded to the egress router to remove the SRv6 encapsulation before sending the packet to its final destination.

3.4.3 Control plane

The architecture of the control plane includes the *End-to-End Orchestrator* that interacts with the *NFV Managers* for configuring and administrating the VNFs. In a simple scenario, VNFs are instantiated and configured in the NFV nodes by the orchestrator in a static way. This means that the VNFS will ready and waiting to process packets steered through them. In a more advanced scenario, VNFs could be instantiated (on-the-fly). This means that the orchestrator will instantiate and configure the VNF upon the arrival of the first packet. This solution is more efficient in terms of resource utilization as VNFs will be deployed only when there is traffic steered through them. However, it introduces some delay as the first few packets need to wait until the VNF get configured. The SDN controller is responsible for configuring the network. In particular, the SDN controller performs the following:

1. Configuration of SRv6 *Ingress* routers with traffic classification rules and the different SRv6 SID lists to be applied to classified traffic. Such approach requires a per-flow configuration state only at the *Ingress* node, and no per-flow states of the rest of the network are needed.
2. Configuration of the routing across all nodes of the SRv6 domain. In the absence of specific requirements for traffic engineering, this function can be replaced by standard routing protocols, like OSPF. This function is not executed on a per-flow basis, since it is requested only when a new node is added or removed, or when some modification of the routing is needed.

3.5 SRv6-unaware VNFs

An SRv6-unaware VNF is not able to process the SRv6 information in the traffic that it receives. It may either drop the traffic or take erroneous decisions due to the unrecognized routing information. In order to include such VNFs in an SRv6 SFC policy, it is thus required to remove the SRv6 information before the VNF receives the packet, or to alter it in such a way that the VNF can correctly process the packet.

The SRv6 proxy is an entity, separate from the VNF, that performs these modifications and handles the SRv6 processing on behalf of a VNF [41]. It can run as a separate process on the VNF appliance, on a virtual switch or router on the NFV node or on a remote host. We assume that the SRv6 proxy is connected to the VNF via a layer-2 link. An SRv6-unaware VNF is associated with an SRv6 SID instantiated on the SRv6 proxy, which is used to steer traffic through the VNF.

In collaboration with several network device vendors and network operators, we designed several SRv6 proxy behaviors to enable SR SFC through SRv6-unaware VNFs [41] [46]. A system implementing one of these functions may handle the SR processing on behalf of an SRv6-unaware VNF and allows the VNF to properly process the traffic that is steered through it. In general, a VNF may be located at any hop in an SR SFC, including the last segment. However, the SRv6 proxy behaviors defined in this section are dedicated to supporting SRv6-unaware VNFs at intermediate hops in the segment list. In case an SRv6-unaware VNF is at the last segment, it is sufficient to ensure that the SR information is ignored (IPv6 routing extension header with Segments Left equal to 0) or removed before the packet reaches the VNF.

The generic behavior of an SRv6 proxy has two parts as illustrated in Figure 3.2. The first part is in charge of passing traffic from the network to the VNF. It intercepts the SR traffic destined for the VNF via a locally instantiated SRv6 SID, modifies it in such a way that it appears as non-SR traffic to the VNF, then sends it out on a given interface, *IFACE-OUT*, connected to the VNF. The second part receives the traffic coming back from the VNF on *IFACE-IN*, restores the SR information and forwards it according to the next segment in the list.

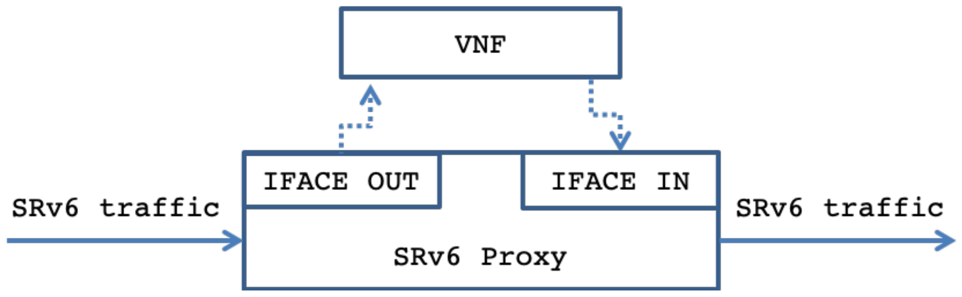


Figure 3.2: SRv6 proxy.

Static, *Dynamic*, *Shared-memory*, and *Masquerading* are different SRv6 proxy mechanisms. Each of which has its own characteristics and constraints. It is up to the operator to select the best one based on the proxy node capabilities, the VNF behavior and the traffic type. It is also possible to use different proxy mechanisms within the same service chain [41]. In the following subsections we will discuss these different proxy mechanisms.

3.5.1 Static proxy

The static proxy is an SRv6 endpoint behavior for processing SRv6 traffic on behalf of an SRv6-unaware VNFs. We consider only the case of encapsulation mode, hence this proxy receives SR traffic that is formed of an outer IPv6 header with SRH, encapsulating an inner packet, which can be Ethernet, IPv4 or IPv6. A static SRv6 proxy segment is associated with the following mandatory parameters:

- *INNER-TYPE*: inner packet type
- *S-ADDR*: ethernet or IP address of the VNF (only for inner type IPv4 and IPv6)
- *IFACE-OUT*: local interface for sending traffic towards the VNF
- *IFACE-IN*: local interface receiving the traffic coming back from the VNF
- *CACHE*: SR information to be attached on the traffic coming back from the VNF.

A static SRv6 proxy segment is thus defined for a specific VNF, inner packet type (*INNER-TYPE*) and cached SR information (*CACHE*). It is also bound to a pair of directed interfaces on the proxy (*IFACE-OUT* and *IFACE-IN*). These may be both directions of a single interface, or opposite directions of two different interfaces. The latter choice is recommended in case the VNF is to be used as part of a bi-directional SR SFC policy.

The first part of this behavior is triggered when the proxy node receives a packet whose active segment matches a segment associated with the static proxy behavior. It removes the SR information from the packet then sends it on a specific interface towards the associated VNF. This SR information corresponds to the encapsulation IPv6 header with any attached extension header in the case of SRv6.

The second part is an inbound policy attached to the proxy interface receiving the traffic returning from the VNF, *IFACE-IN*. This policy attaches the cached SR information associated with the SRv6 proxy segment to the incoming traffic. The *CACHE* is defined as a source address, an active segment and an optional SRH (tag, segments left, segment list and metadata). The proxy encapsulates the packets with an IPv6 header that has the source address, the active segment as destination address and the SRH as a routing extension header. After the SR information has been attached, the packets are forwarded according to the active segment, which is represented in the IPv6 destination address.

In this scenario, there are no restrictions on the operations that can be performed by the VNF on the stream of packets. It may operate at all protocol layers, terminate transport layer connections, generate new packets and initiate transport layer connections. This behavior may also be used to integrate an IPv4-only service into an SRv6 policy. However, a static SRv6 proxy segment can be used in only one service chain at a time.

As opposed to most other segment types, a static SRv6 proxy segment is bound to a unique list of segments, which represents a directed SR SFC policy. This is due to cached SR information being defined in the segment configuration. This limitation prevents multiple segment lists from using the same static SRv6 proxy segment at the same time. Note that the same segment list can be shared among mutliple traffic flows. Besides, since the returning traffic from the VNF is re-classified based on the incoming interface, an interface can be used as receiving interface *IFACE-IN* only for a single SRv6 proxy segment at a time. In the case of a bi-directional SR SC policy, a different SRv6 proxy segment and receiving interface are required for the return direction.

3.5.2 Dynamic proxy

The dynamic proxy is an improvement over the static proxy that dynamically learns SR information before removing it from the incoming traffic. The same information can then be re-attached to the traffic returning from the VNF. As opposed to the static SRv6 proxy, no *CACHE* information needs to be configured. Instead, the dynamic SRv6 proxy relies on a local caching mechanism on the node instantiating this segment. Therefore, a dynamic proxy segment cannot be the last segment in an SR SFC policy.

Upon receiving a packet whose active segment matches a dynamic SRv6 proxy function, the proxy node applies the SRv6 End behavior, then compares the updated SR information with the cache entry for the current segment. If the cache is empty or different, it is updated with the new SR information. SR information is then removed and the inner packet is sent towards the VNF. The cache entry is not mapped to any particular packet, but instead to an SR SFC policy identified by the receiving interface *IFACE-IN*. Any non-link-local IP packet or non-local Ethernet frame received on that interface will be re-encapsulated with the cached headers. The VNF may thus drop, modify or generate new packets without affecting the proxy.

3.5.3 Shared-memory proxy

The shared memory proxy is an SRv6 endpoint behavior for processing SRv6-encapsulated traffic on behalf of an SRv6-unaware VNF. This proxy behavior leverages a shared-memory interface with the service in the VNF in order to hide SR information from an SRv6-unaware VNF while keeping it attached to the packet. We assume in this case that the proxy and the VNF are running on the same NFV node.

3.5.4 Masquerading proxy

The masquerading proxy is another SRv6 endpoint behavior for processing SRv6 traffic on behalf of an SRv6-unaware VNF. This proxy receives SR traffic that is formed of an IPv6 header and an SRH on top of an inner payload. The masquerading behavior is independent from the inner payload type. Hence, the inner payload can be of any type but often in practice a transport layer packet, such as TCP or UDP. A masquerading SRv6 proxy segment is associated with the following mandatory parameters:

- *S-ADDR*: ethernet or IPv6 address of the VNF
- *IFACE-OUT*: local interface for sending traffic towards the VNF
- *IFACE-IN*: local interface receiving the traffic coming back from the VNF.

A masquerading SRv6 proxy segment is thus defined for a specific service and bound to a pair of directed interfaces or sub-interfaces on the proxy. As opposed to the dynamic SR proxies, a masquerading segment can be present at the same time in any number of SR SFC policies and the same interfaces can be bound to multiple masquerading proxy segments. The only restriction is that a masquerading proxy segment cannot be the last segment in an SR SFC policy.

The first part of the masquerading behavior is triggered when the proxy node receives an IPv6 packet whose destination address matches a masquerading proxy segment. The proxy inspects the IPv6 extension headers and substitutes the destination address with the last segment in the SRH attached to the IPv6 header, which represents the final destination of the IPv6 packet. The packet is then sent out towards the VNF. The VNF receives an IPv6 packet whose source and destination addresses are respectively the original source and final destination. It does not attempt to inspect the SRH, as RFC8200 [8] specifies that routing extension headers are not examined or processed by transit nodes. Instead, the VNF simply forwards the packet based on its current destination address. In this scenario, we assume that the VNF can only inspect, drop or perform limited changes to the packets. For example, intrusion detection system (IDS), deep packet inspector (DPI) and non-NAT firewalls are among the services that can be supported by a masquerading SRv6 proxy.

The second part of the masquerading behavior, also called de-masquerading, is an inbound policy attached to the proxy interface receiving the traffic returning from the VNF, *IFACE-IN*.

This policy inspects the incoming traffic and triggers a regular SRv6 endpoint processing on any IPv6 packet that contains an SRH. This processing occurs before any lookup on the packet destination address and it is sufficient to restore the right active segment as the destination address of the IPv6 packet.

3.5.5 Packet processing

As shown in Figure 3.1, the SRv6 proxy is running inside each NFV node to handle the processing of SR information on behalf on SRv6-unaware VNF. This operation can be logically split into three phases:

1. identifying the target VNF and modifying the packet.
2. forwarding the packet to the correct VNF
3. restoring the correct SRv6 encapsulation after the packet being processed by the VNF.

In this section we explain in detail the process of including SRv6-unaware VNFs in SRv6-based SFC. As an SRv6 proxy we use the SR dynamic proxy, since this proxy can support a broad range of use-cases. However, most the concepts discussed in this section also apply to other proxy behaviors. When an SR-encapsulated packet arrives to an NFV node with its destination address matching an SRv6-unaware VNF running on that NFV node, the packet is handed the dynamic SRv6 proxy. Then following operations are performed:

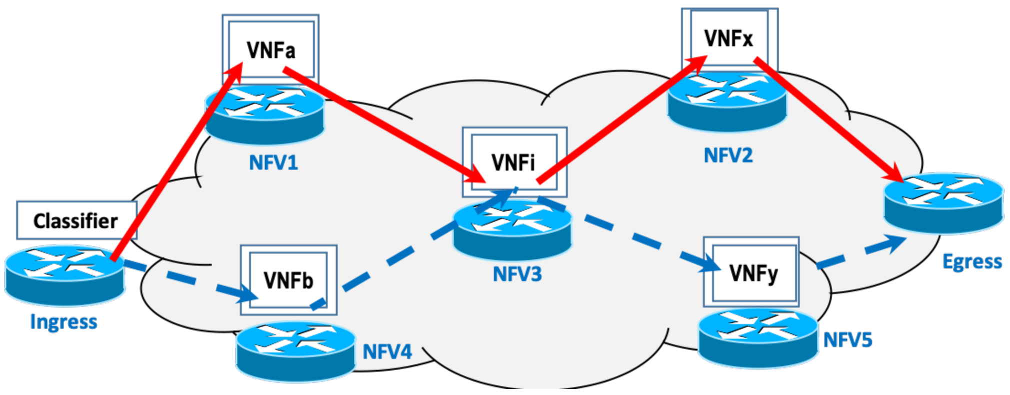
- the proxy extracts the SRv6 encapsulation from the packet, including the outer IPv6 header, SRH, and any other IPv6 extension header
- the proxy maintains a *CACHE* for previously saved SRv6 encapsulations, and compares the extracted SRv6 encapsulation to the *CACHE* elements.
- if the *CACHE* is empty or the SRv6 encapsulation is different from the cached one, then the *CACHE* is updated with the new SRv6 encapsulation
- the original packet is extracted and forwarded to the SRv6-unaware VNF through the *IFACE-OUT* interface
- the VNF processes the packet; there is no restriction on the processing done by the VNF, hence the packet can be modified, forwarded, or deleted in the VNF

- packets processed by the VNF are sent back to the proxy through the *IFACE-IN* interface
- the proxy searches its *CACHE* using the *IFACE-IN* as a key
- the cached SRv6 encapsulation is retrieved and re-added to packets
- finally the packet is handed back to the routing system of the NFV node to be forwarded to the next VNF in the chain, which might be running on the same NFV node or on a different one.

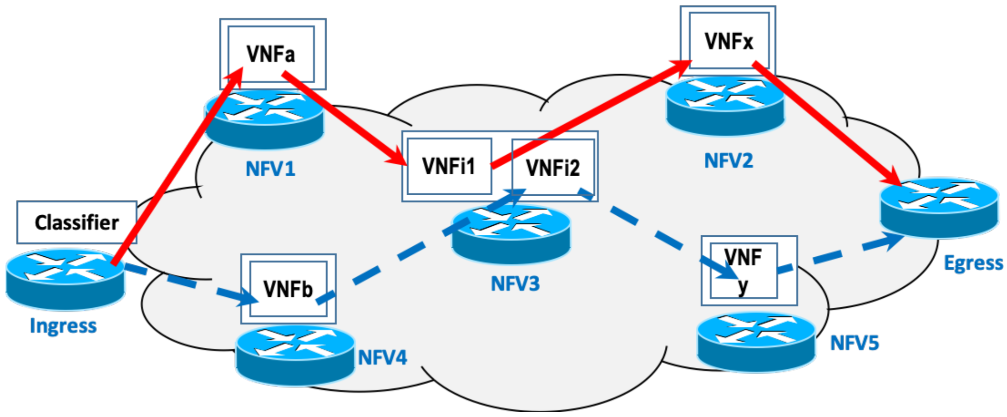
Generally, a VNF could be part of several VNF chains at the same time. Therefore, packets that go through a VNF should be re-classified after being processed to identify the chain they belong to. On the other hand, we can impose the constraint that a VNF can be part of only chain at the same time. Under this constraint, it is possible to operate in a very simple way and associate all packets processed by the VNF to one VNF chain. In Figure 3.3, we assume that packets belonging to a flow f_1 are associated by an ingress node to the VNF chain represented by <VNFa,VNFi,VNFx>, while the packets belonging to the flow f_2 are associated to the chain <VNFb,VNFi,VNFy>. The packets of both flows need to cross the VNF $VNFi$, but those belonging the the flow f_1 should be associated to the chain <VNFa,VNFi,VNFx> when they go out of the VNFi and those belonging to the flow f_2 should be associated to the other chain. This scenario is shown in Figure 3.3(a). If we duplicate the VNFi by instantiating two instances in the same VNF node ($VNFi1$ and $VNFi2$ in Figure 3.3(b)), it is possible to meet the condition that a VNF is part of only one chain. The two chains associated to flows f_1 and f_2 will now be <VNFa,VNFi1,VNFx> and <VNFb,VNFi2,VNFy> respectively.

In our architecture we assume that VNFs are univocally mappable, i.e., they belong to at most one VNF chain. We also assume that a VNF cannot appear twice in the same VNF chain. A bi-directional service chain is composed of two uni-directional chains, indicated as Eastbound chain and Westbound chain in Figure 3.4. We assume that each VNF participating in a bi-directional service chain has two interfaces, indicated as W and E. Under these assumption, outgoing traffic from the E interfaces can be associated to the Eastbound chain (from source node S to dest node D) while traffic going out from the W interfaces can be associated to the Westbound chain (from D to S). In other words, we assume that each interface of a VNF instance can inject traffic only in one chain at a time.

Formally, let S be the set of all VNF chains allocated in the network. Let I be an interface of a VNF instance. A *univocally mappable* interface is an interface that has all the traffic outgoing from I belongs to at most one VNF chain in S. For bi-directional chains, a *univocally mappable*

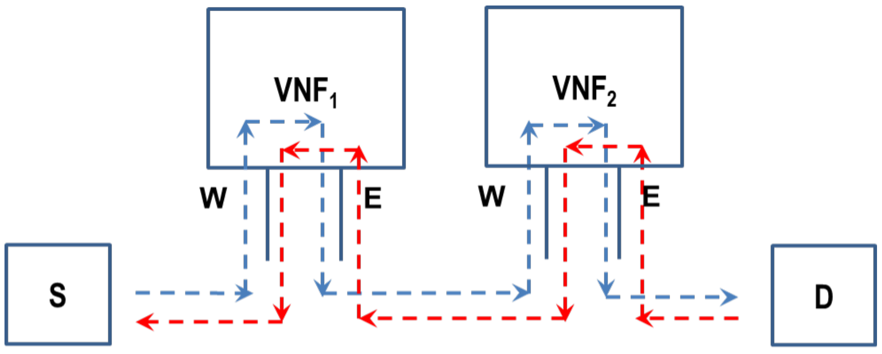


(a) **VNF as part of several chains**



(b) **VNF as part of a single chain only**

Figure 3.3: Service Function Chains examples



Eastbound chain: $\langle$ VNF_1-W, VNF_2-W, D $\rangle$

Westbound chain: $\langle$ VNF_2-E, VNF_1-E, S $\rangle$

Figure 3.4: Bi-directional Service Chain example

VNF is a VNF that has all its interfaces that output traffic are univocally mappable. In case of non-univocally mappable VNFs, a more complex classifier is needed in the SRv6 proxy to associate the packets outgoing from the VNF to the VNF chain and index. This case is not covered in this the thesis.

3.6 Implementation

The SRv6-based SFC architecture we have shown in Figure 3.1 is composed of several components; *Ingress* node, *Egress* node, classifier, NFV nodes, VNFs, proxy, and the control plane. We chose Linux to implement our SFC architecture because of its feature richness which makes it an ideal platform to implement the various components of the architecture. Linux is commonly used within the research community and thus gives more opportunities for collaboration and eases the process of re-producing our work by other researchers. In addition, by using Linux we can leverage the various SRv6 capabilities supported in kernel 4.10 and 4.14. An alternate would have been VPP [21]. However, this would require special hardware (e.g. NIC) that supports DPDK [47]. Moreover, VPP would have made it more difficult for other researchers to collaborate or reproduce our work.

We implemented all the components of our SFC architecture, except the SRv6 proxy, using the native features of the Linux kernel. To implement the SRv6 proxy in Linux we chose between two different implementations choices: integrate the proxy implementation directly in the kernel, or implement it as an external kernel module. In the first phase, we preferred to use the latter modular approach and implemented a new kernel module called SREXT. This approach gives us more flexibility as it is possible to update the implementation without recompiling the kernel each time. It also eases the process of installing our extension on any Linux machine. The SREXT implementation is open source and publicly available [48]. We also considered the other choice (integrate the proxy implementation directly in the kernel) and contributed to the design of a native kernel SR-Proxy (SRNK) which we cover in Section 3.6.2.

3.6.1 SREXT

SREXT is a kernel module providing the basic segment routing functions in addition to more advanced ones. It can be used as a standalone SRv6 implementation or as a complement to the existing SRv6 kernel implementation (kernel 4.10 and later kernels). SREXT supports a "localsid table" which contains the local SRv6 segments explicitly instantiated in the node and associates each SID with a function. The localsid table is completely independent from the Linux routing table and contains SRv6 SIDs only. Each entry of the localsid table is an SRv6 segment that is associated with an SRv6 endpoint behavior.

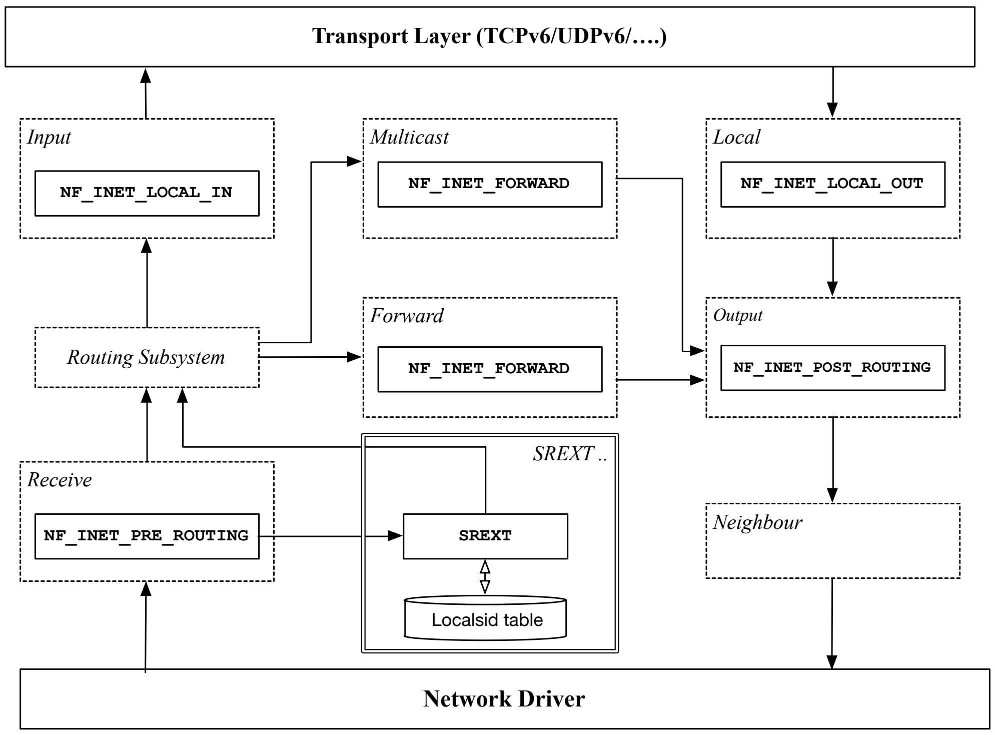


Figure 3.5: SREXT Architecture

SREXT registers a callback function in the pre-routing hook of the netfilter framework as shown Figure 3.5. This callback function is invoked for each received IPv6 packet. If the destination address of the IPv6 packet matches an entry in the localsid table, the associated behavior is applied, otherwise the packet will go through the kernel routing sub-system for normal processing. SREXT removes the SRv6 encapsulation from packets and hands only the inner packet to the VNF. Before handing packets to the VNF, SREXT creates a cache entry with the removed encapsulation and uses the interface connecting to the VNF as a key. As SREXT is registered in the pre-routing hook, it receives packets before the routing sub-system. Hence, it will be able to re-apply the cached SRv6 encapsulation to packets coming back from VNF. The currently supported SRv6 endpoint behaviors by SREXT are listed in Table 3.1.

Behavior	Description
End	*Endpoint function is the mostbasic function*
End.X	*Endpoint with cross-connect to an array of L3 adjacencies*
End.DX2	*Endpoint with decapsulation and L2 cross-connect to OIF*
End.Dx4	*Endpoint with decapsulation and cross-connect to an IPv4 adjacency*
End.DX6	*Endpoint with decapsulation and cross-connect to an IPv6 adjacency*
End.AD4	*Endpoint to IPv4 SR-unaware APP via dynamic proxy*
End.AD6	*Endpoint to IPv6 SR-unaware APP via dynamic proxy*
End.AM	*Endpoint to SR-unaware APP via masquerading*
End.EAD4	*Extended End.AD4 to allow SFs be the last SID*
End.EAD6	*Extended End.AD6 to allow SFs be the last SID*

Table 3.1: SREXT Supported SRv6 behaviors

We provided a native command-line-interface (CLI), named *srconf*, that allows to add, delete, show, flush or clear counters of localsid table entries as shown in Listing 3.1. We also provide a simple and easily replicable testbed to experiment with our implementation. The testbed is composed of three nodes representing SRv6 *Ingress*, *NFV* and *Egress* nodes. The testbed is based on VirtualBox [49] and Vagrant [50]. VirtualBox is a virtualization hyper-visor used to create virtual machines (VMs). Vagrant is an open source tool for creating portable VMs. It simplifies the life cycle management and provisioning of VMs. A full description and instructions to replicate the testbed are available at [48].

SREXT has been widely adopted by the research community as well as industry as an SRv6 proxy implementation in Linux [51] [39] [52] [34]. It was included in the SRv6 interoperability demo which was presented the SIGCOMM 2017 Industrial demos. The demo shows the interoperability among hardware platforms from several network vendors as well as open source SRv6 implementations both in the Linux kernel and VPP. It showcases two main use-cases of SRv6: layer-3 VPN and service chaining. SREXT allowed to include some SRv6 unaware VNFs in the service chain.

Listing 3.1: SREXT Command Line Interface (CLI)

```
$ sudo srconf localsid
Usage: srconf localsid { help | flush }
       srconf localsid { show | clear-counters } [SID]
       srconf localsid del SID
       srconf localsid add SID BEHAVIOR
BEHAVIOR:= { end |
              end.dx2 OIF |
              end.dx4 NH4 OIF |
              { end.x | end.dx6 } NH6 OIF |
              { end.ad4 | end.ead4 } NH4 OIF IIF |
              { end.am | end.ad6 | end.ead6} NH6 OIF IIF}
NH4:= { ip IPv4-ADDR | mac MAC-ADDR }
NH6:= { ip IPv6-ADDR | mac MAC-ADDR }
```

3.6.2 Integration in the Linux kernel mainline

SRNK (SR-Proxy Native Kernel) is a proposal to re-implement SREXT as part of the Linux kernel mainline [38]. SREXT was implemented to complement the Linux kernel to support the SRv6 behaviors that we were not supported yet. By that time, the Linux kernel was only supporting the basic SRv6 *End* behavior processing. Currently, most of the behaviors implemented in SREXT are supported by the mainline of Linux kernel, except the SRv6-proxy behaviors. While the SREXT implementation has been acknowledged by the research community and the industry, SRNK proposes a mechanism to integrate the implementation of the static and dynamic proxy behaviors directly in the kernel. We contributed to the design and evaluation of SRNK [38] [53], which is implemented by Netgroup Research Group from University of Rome Tor Vergata [54] [55].

3.7 Related work

Network Service Header (NSH) provides an alternative solution to implement SFC [56]. The idea is to carry both SFC metadata and path information in an additional header. The traffic steering between VNFs is delegated to other tunneling mechanisms such as VXLAN [44] or GRE [43]. Traffic is encapsulated in tunnels that go from one VNF to the next VNF hop, the NSH is added between the tunnel headers and the original packet. The basic mechanism is designed to work with NSH-aware services that are capable to understand and process NSH headers. The largest majority of VNF legacy services is NSH-unaware, in this case before the traffic reaches and after it leaves the VNF a NSH proxy has to process it in order to remove and then reattach the NSH header. Such type of proxy has to remove and reattach the NSH header to each incoming/outgoing packet from the VNF, hence it has to reclassify all the outgoing packets from the VNF.

The SRv6-based approach and the NSH approach are not necessarily mutually exclusive, but there can be coexistence scenarios. In particular, citing [13], “the use of SRv6 together with the NSH allows building flexible service chains where the topological information related to the path to be followed is carried into the Segment List while the *service plane related information* (function/action to be performed) is encoded in the metadata, carried into the NSH”.

NSH is supported in Open vSwitch [57] through an unofficial patch [58]. The same patch is used in [59] to offer experimental support to NSH in the OpenDaylight [28] platform. OpenStack Neutron [60] is the networking-as-a-service platform used by the OpenStack [61] project. Currently Neutron does not support service function chaining. There is a proposed SFC API for OpenStack Neutron [62]. This API defines a service chain as: i) flow classifier - definition of what traffic enters the chain; ii) an ordered list of Neutron ports that define the chain; iii) correlation type - chain metadata encapsulation type. VMs are connected to a Neutron network via Neutron ports, using an ordered chain of ports the traffic is steered through the VMs composing the service chain. This makes it possible to create a traffic steering model for service chaining that uses only Neutron ports. This traffic steering model has no notion of the actual services attached to these Neutron ports. The *correlation type* specifies the type of chain correlation mechanism supported by a specific service functions (it can be MPLS, NSH, ecc..). This is needed, in case of SFC-aware SFs, by the data plane switch to determine how to associate a packet with a chain. In case of SFC-unaware VNFs (VNFs that do not support any correlation mechanism) the correlation type is set to none.

The SRv6 solution for SFC still has several advantages compared to NSH. In particular, it provides a unique solution for service function chaining combined with overlay networking and tight underlay requirements such as low latency and high bandwidth. The SRv6 solution can also scale better, as the SFC state is maintained only at the SRv6 *Ingress* router with no need for state information in the network fabric.

Chapter 4

SRv6-aware network functions

4.1 Introduction

SRv6-aware NFs are capable of processing SRv6 information in the packets. They can be integrated into SRv6 service chains with no need of any sort of proxy behavior. This simplifies the configuration and management of the NFV infrastructure. At the same time this has a positive effect on the performance of the NFV-enabled nodes, as there is no complex classification procedures required anymore. Moreover, it allows to implement advanced SFC features, as the SRH information is preserved when packets are processed by the NF. In this chapter, we take a forward-looking approach and consider the design and implementation of SRv6-aware network functions. We provide open source implementations for these SRv6-aware NFs that has been partly adopted in the mainline of the Linux kernel.

The content of this chapter has been partially or fully published in [46] [40] [63] [39]. In Section 4.2, we define the design considerations for developing SRv6-aware NFs. In Section 4.3-4.6 we explain the design and implementation of several SRv6-aware NFs. In Section 4.3 we provide a full description of the design, implementation and performance of our SRv6-aware firewall, named SERA. In Section 4.4 we explain how we developed an SRv6-aware IDS (Intrusion Detection System) and IPS (Intrusion Prevention System) by extending the widely used open source Snort [64]. In Section 4.5 we describe the the SRv6 extension that has been added to Linux *nftables* firewall [65]. Finally, in Section 4.6 we augment TCPDUMP [66] with SRv6 capabilities for debugging SRv6 traffic.

4.2 Designing SRv6-aware NFs

SRv6-aware NFs are capable of processing SRv6-encapsulated traffic, which means that they can process the original packet despite the fact that it has been encapsulated with SRv6 encapsulation. They can identify the sequence of NFs that have already processed the packet and the sequence of following NFs that still have to process the packet. They can also modify the service chain by adding, removing, and possibly reordering the segment list. This opens the possibility of advanced service chaining operations, such as branching and looping. When a node running SRv6-aware NFs receives an SRv6-encapsulated packet whose destination address matches a segment assigned to an SRv6-aware NF, the full SRv6 packet is processed to the VNF. The VNF can act on packet's outer IPv6 header, SRH and inner packet. It can perform any action to modify the packet or even drop it. The VNF can modify the packet path by adding or removing a SID to the SID list carried in the SRH. Finally, it updates the packet destination address with the next SID from the SID list and forwards the packet accordingly.

In this section, we choose the firewall as an example to define the design considerations for SRv6-aware NFs. However, these considerations have a general value as they can be applied to several types of network functions that need to be deployed on an SRv6-based SFC environment (e.g. DPI, IDS). A firewall essentially works according to a set of rules to accept or drop received packets [67]. Each rule is composed of a condition and an action. The condition is based on the attributes of the received packets. Once a packet satisfies the condition expressed by a rule condition, the associated action is performed on that packet

We assume that an SRv6-aware firewall should support two working modes: *basic* mode and *advanced* mode. In the basic mode the SRv6-aware firewall must be able to work as a legacy firewall, but with no need of the SRv6-proxy. In particular, the SRv6-aware firewall should be able to use the same set of rules defined for the legacy firewall and apply them directly to the SRv6-encapsulated packets that carry SRH information. It must be able to handle SRv6 packets encapsulated in *encap* as well as *insert* modes and logically apply the rules to the original packets rather than to the SFC encapsulated packets. To make a concrete example, if an existing rule includes a condition on the source IPv6 address and the original IPv6 packet has been encapsulated using IPv6-in-IPv6 it makes no sense to consider the outer IPv6 source address of the received packet as the condition should be checked on the source address of the original packet. The use-case scenario is to virtualize the legacy firewalls, executing them in servers on the NFV infrastructure, without changing the legacy rules and with no need of SRv6-proxy functionality.

In the advanced mode the SRv6-aware firewall should support rules with extended conditions that can explicitly include attributes not only from the original packet but also from the SRH and the outer packet. In particular, the SRv6-aware firewall could leverage SRv6 SID arguments, TLVs, or TAG. It could also apply differentiated processing based on the active SRv6 SID (i.e., apply different rule sets for different SIDs). As for the actions, in the advanced mode the SRv6-aware firewall should be able to support SRv6-specific actions. For example, an SRv6-specific action could be to skip the next SID in the segment list, so that it is possible to operate a "branching" instead of the usual linear exploration of the VNF chain, when some conditions on the packet are met. A use-case scenario for this feature is a service chain which includes a firewall followed by an Intrusion Detection System (IDS) and allow skipping the IDS for a subset of traffic that matches some conditions.

A further requirement is that the SRv6-aware firewall application should be able to select the actions to be performed based on information contained in the SID. This is aligned with the SRv6 network programming approach [34] of minimizing the state information maintained in the nodes and storing explicit state information in the packets. The use-case scenario in this case is that instead of re-configuring some firewall rules in a specific firewall running in the core of the NFV infrastructure, it is possible to obtain the same result by changing a SID in the SID list that is injected to the packet by the edge node. The big advantage is that the reconfiguration is only needed at the edge node.

4.3 SERA (SEgment Routing Aware firewall)

SERA (SEgment Routing Aware firewall) is an advanced SRv6-aware firewall, capable of taking stateless actions programmed in the SRH. It extends the Linux iptables firewall and supports both *basic* and *advanced* modes of SRv6-aware firewalls. The *basic* mode of the proposed SERA firewall solution avoids the need of (re)classifying packets in the intermediate NFV nodes that host the SRv6-aware firewall. The *advanced* mode supports new firewall actions that can operate on the SRH segment list, allowing to make branches in the VNF chain. To the best of our knowledge, the SERA firewall can be considered the first SRv6-aware application.

4.3.1 SERA basic mode

In the basic mode, SERA applies the firewall processing to the original packets of SRv6 traffic. The proposed packet processing architecture is shown in Figure 4.1. Each received packet goes through an *SRv6 pre-processor* that splits traffic into SRv6 and non-SRv6 traffic. Non-SRv6 traffic does not require any special processing and is processed as in an SRv6-unaware firewall, as represented with the solid-line path in Figure 4.1. SRv6 traffic follows a different path through the firewall, represented with double-line path in the figure. In this path, the firewall, using the *Inner match* module, evaluates the rules on the inner packet, properly taking into account the impact of the SRv6 encapsulation. The *Inner match* module supports both *encap* and *insert* modes of SRv6, which implies that the original IPv6 source and destination information of received packets may be encoded differently. In the *encap* mode the original source and destination are the ones of the original packet. In the *insert* mode, packets have only one IPv6 header. The original source information is in the source address of the IPv6 header, while the original destination is encoded as the last SID in the SRH. The *Inner match* module is responsible for getting the original source and destination information from SRv6 packets and comparing them to the defined rules. Once a packet matches one of the rules, the *Action* module applies the associated action (e.g., ACCEPT, DROP) on that packet.

4.3.2 SERA advanced mode

In the advanced mode, SERA extends the iptables capabilities with new matching capabilities and new SRv6-specific actions. It introduces a new type of iptables rules (SERA rules) that have extended conditions on the attributes of outer packet, inner packet, and the SRH header. The architecture of advanced mode (Figure 4.2) is defined incrementally with respect to the basic mode (Figure 4.1), by adding the *SRH match* module and replacing the *Action* block with the *Extended Action* block. Since the matching could be performed on both the original and the outer packet headers, the SRv6 traffic follows a more complex path, as shown in Figure 4.2. Unlike in the basic mode SERA, all received packet are first processed by the *Outer match* block, which applies parts of the extended rules on the outer packet.

The *SRv6 pre-processor* does the same job as in the basic mode SERA by splitting traffic into non-SRv6 and SRv6 traffic. Non-SRv6 traffic goes directly to the *Action* module while SRv6 traffic is directed to the *Inner match* module. The *Inner match* module works as in the basic mode, but the rules that drive its behavior are written in a different way. For example,

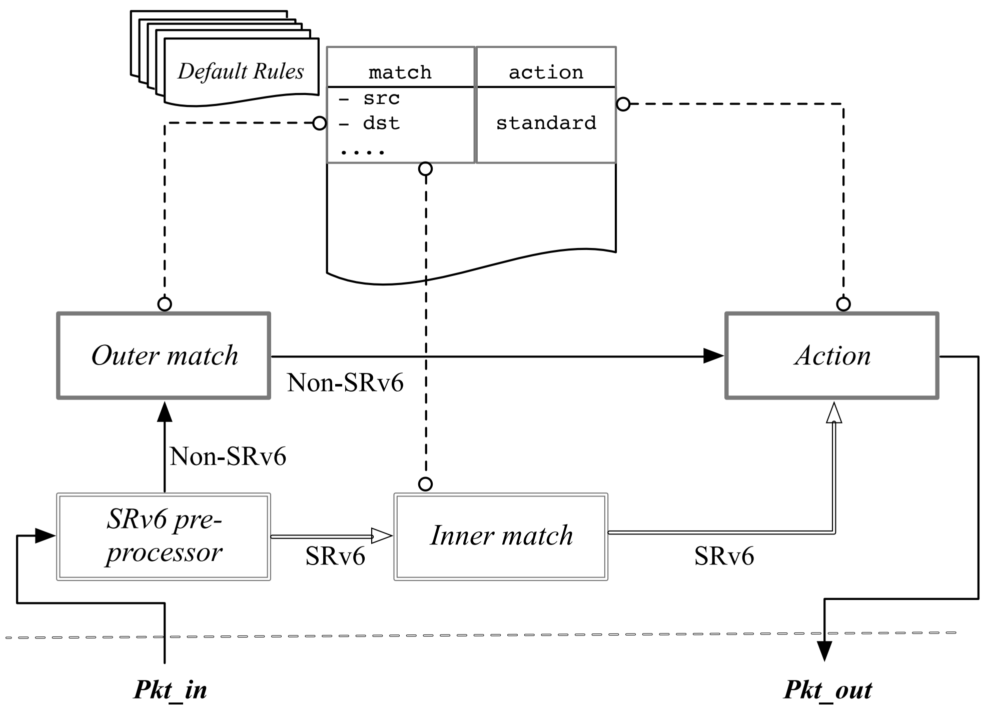


Figure 4.1: SERA Architecture: basic mode

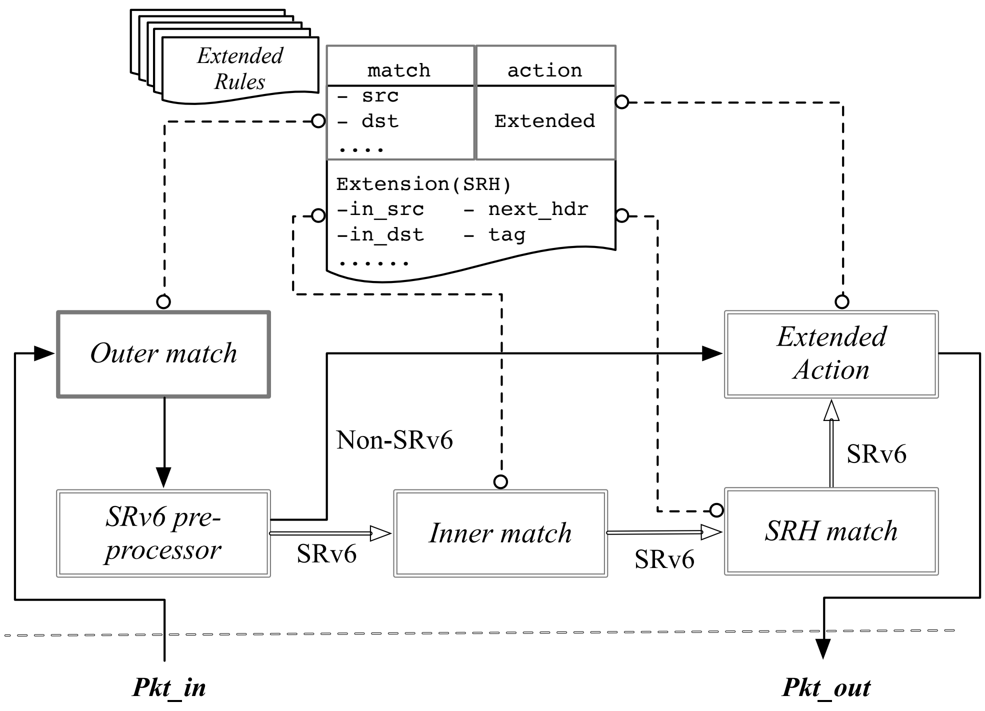


Figure 4.2: SERA Architecture: advanced mode

with an extended rule it is possible to match on the outer source and destination IPv6 addresses (denoted as *src*, *dst*) and on the original ones (denoted as *inner-src*, *inner-dst*). The *Inner match* block takes care of the matching of the inner source and destination (the ones of the original packet). The *SRH match* block is concerned with the matching between SRH extension part of the rules and the SRH of received SRv6 packets. Finally, each packet (SRv6 or non-SR) that satisfies the matching condition of a rule goes to the *Extended Action* module. It extends the Action module present in the architecture of the Basic mode by allowing the introduction of *SRv6-specific* actions in addition to the standard ones.

An SRv6-specific action is an advanced action that can be applied to SRv6-encapsulated packets. It may modify or process SRv6-encapsulated packets based on SRH information. We list here some examples of SRv6-specific actions, but the set of these actions can be extended to cover more complex SFC use-cases.

- *seg6-go-next*: Similar to the SRv6 *End* behavior from the SRv6 network programming model [34]. It sends packets towards the next SID from SRH. The *seg6-go-next* serves as an ACCEPT action for SRv6 encapsulated packets.
- *seg6-skip-next*: Instructs the SERA firewall to skip the next SID in the SRH.
- *seg6-go-last*: Instructs the SERA firewall to skip the remaining part of the segment list and process the last segment.
- *seg6-eval-args*: Generic action to support programming actions into the SRH content. (more below).

Following the traditional iptables model, the above defined SRv6-specific actions are included in *statically* configured rules which are executed in a SERA firewall running as a VNF. Taking into account the concepts of the SRv6 programming model, we have designed a more dynamic approach, which allows defining the action to be executed as a result of a match on a packet-by-packet basis, by putting information in the SID. For this purpose, the special SRv6-specific action (*seg6-eval-args*) is defined. It does not represent a concrete action, but it instructs the SERA firewall to look into the current SID to find the action to be executed. In order to understand how this dynamic approach works, let us recall first how the SRv6 SIDs are structured. The SRv6 network programming model defines SIDs as IPv6 addresses which can be logically split into three fields *LOC*:*FUNCT*:*ARGS* [34]. *LOC* uses the L most significant bits, *ARGS* the R rightmost-bits and *FUNCT* the remaining $128-(L+R)$ bits in the middle.

In our case, the *LOC* part is used as a locator to forward the packets to the NFV node that runs the firewall, and it is advertised by the routing protocols. The *FUNCT* part identifies a specific VNF on the NFV node (in our case the SERA firewall instance). The *ARGS* part may contain information required by the VNF and may even change on a per-packet basis. Note that the *ARGS* part will be ignored in most cases (or omitted setting $R = 0$), whenever there is no need to carry additional information in the SID. For example, the *LOC* field can be 64 bits long and uniquely identify an NFV node. This leaves $128 - 64 = 64$ bits for the identification of the VNF in the NFV node and for the arguments if needed.

In the advanced mode of SERA it is possible to use the *ARGS* part of the SID to encode a firewall action to be executed in case of match. This requires that a set of rules with action *eval-args* is configured in the SERA firewall. For all packets that match one of these rules, the action to be executed is contained in the *ARGS* field of the SID. To give an example, the *ARGS* part could be defined as follows: *ARGS* = 0 accept packet, *ARGS* = 1 drop packet. The advantage of this approach is that it is possible to (re)configure the action to be executed on a given subset of packets by operating at the network edge, with no need to update the configuration of the SERA firewall instance running in the core of the NFV infrastructure.

4.3.3 SERA Implementation

We implemented SERA as an extension of the Linux iptables firewall. Before going into the details of the implementation we provide a short tutorial on iptables and netfilter. Then we explain our open source implementation of the SERA firewall [68]. We contributed part of the SERA implementation to the Linux kernel which was merged into release 4.16 of the Linux kernel [69]. We also contributed part of the SERA firewall to the *netfilter.org* project to extend the iptables user-space utility to support new match options merged into release 1.6.2 of iptables [70].

Linux iptables firewall

Iptables is a flexible and modular firewall and it is a standard component of most Linux distributions. It is built on top on the netfilter framework as shown in Fig. 4.3.

The netfilter framework (Appendix A) is a set of hooks in the packet traversal through the Linux protocol stack, which allows access to packets at different points [71]. The current netfilter implementation provides five different hooks (*PREROUTING*, *INPUT*, *FORWARD*, *OUTPUT*, *POSTROUTING*) distributed along the receive and transmit path of packets. Kernel modules can register callback functions at any of these hooks. A callback function, after processing a packet, returns to the netfilter hook the action to be taken on the packet, such as DROP, ACCEPT, QUEUE.

Iptables represents the userspace implementation which allows access to the kernel-level netfilter framework hooks. It defines a set of rules that instruct the kernel what to do with packets coming to or traversing the protocol stack. The implementation of netfilter includes some predefined tables, as shown in Figure 4.3. Each table has a set of chains where iptables rules can be inserted. The currently supported tables are:

- Filter: the default table, it contains rules that are used to filter IP packets
- Nat: mainly used to re-write the source and/or destination addresses of IP packets
- Mangle: a specialized table for mangling packet as they go through the kernel

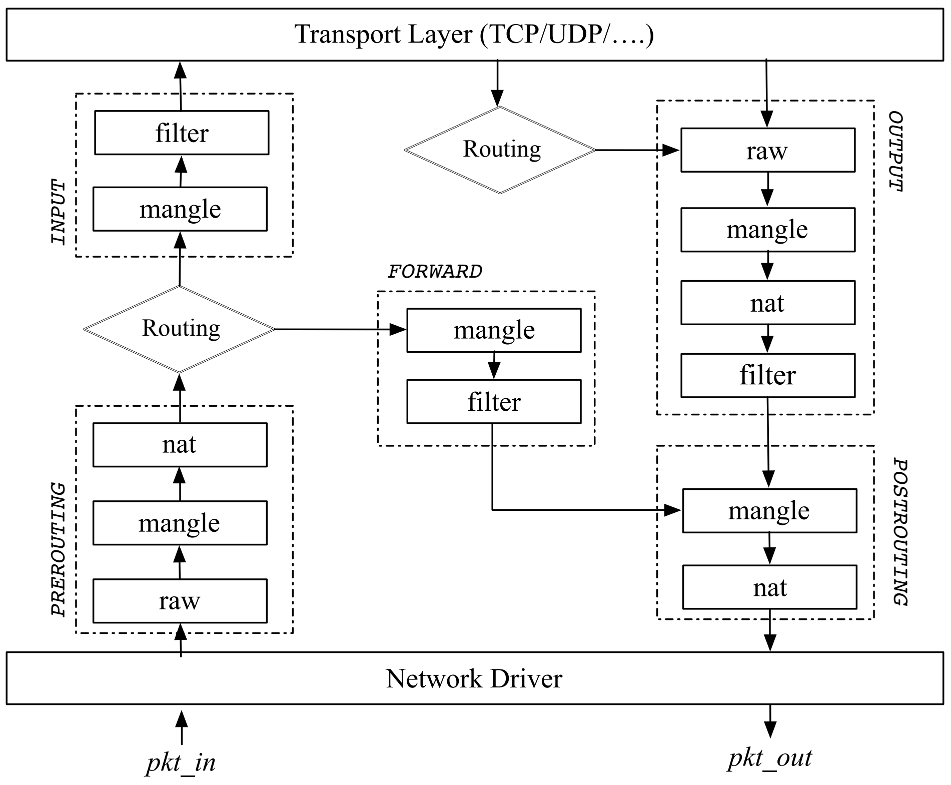


Figure 4.3: Netfilter hooks and their associated tables

- Raw: mainly used for connection tracking.

Each iptables rule defines a set of matching criteria based on information from different layers of the protocol stack. Once a packet matches a rule, iptables takes an action on that packet. The standard actions are: ACCEPT, DROP, or QUEUE. Those correspond to the callback functions return values. Listing 4.1 shows an example of standard iptables rule that matches the packet's source and destination addresses.

The iptables framework is modular and extensible. New match extensions and target extensions can be developed separately and added to the iptables as new modules. Match extensions are used to add more matching options to iptables. They can be used alone or in combination with the default match options. They provide the ability to have sophisticated iptables rules in order to look deeper into IP packets. For instance, hbh matches the parameters in IPv6 Hop-by-Hop extensions header. An example of extended iptables rule is shown in Listing 4.2.

Target extensions are new actions added to the default ones of iptables. A new iptables target usually performs an action different from the default ones (ACCEPT, DROP, etc.,). It can be used for logging/profiling or it can modify the packet before returning it back to the netfilter framework. Destination NAT (DNAT) is an example of iptables target extension, which is used to modify the destination address of a packet.

Implementation of SERA basic mode

In the Linux kernel, the ip6_tables module is responsible for checking the iptables rules against the received packets. It implements the ip6_packet_match() function that evaluates

Listing 4.1: Standard iptables rule

```
# Adding standard iptables rule
$ ip6tables -I INPUT -s fc00:1::/64 -d fc00:d1::/64 \
  -j DROP
```

Listing 4.2: Extended iptables rule

```
# Adding extended iptables rule
$ ip6tables -I INPUT -s fc00:1::/64 -d fc00:d1::/64 \
  -m hbh --hbh-len 40 -j DROP
```

the defined iptables rules against the outermost IPv6 header of a received packet. In order to implement the basic mode of the SERA firewall, we extended the existing ip6_tables module to operate according to the architecture shown in Figure 4.1.

We added the *SRv6 pre-processor* block. SRv6 packets are forwarded to the *Inner match* functional block, implemented by the inner_match() function, which evaluates iptables rules against the original packet. It supports SRv6 packets encapsulated in both encap and insert mode. We added a new *sysctl* parameter (ip6t_seg6) to switch between legacy iptables mode and SERA basic mode. The system administrator can enable the SERA basic mode on the fly with the command shown in Listing 4.3, which activates the *SRv6 pre-processor*. We have implemented a first version of basic mode SERA that implements a subset of the normal classification rules, namely those involving the IP source and destination addresses. On this first version we have performed the evaluation that is reported in Section 4.3.4. Then we have implemented a second version that supports all the classification rules and it is now available at [69].

Listing 4.3: Linux CLI for SERA basic mode.

```
# Enable SERA basic mode.
$ sysctl -w net.ipv6.ip6t_seg6=1
```

Implementation of SERA advanced mode

We implemented the advanced mode SERA by exploiting the iptables extension features. We added a new match extension as well as a new target extension to the iptables implementation both at kernel and user-space levels. Thanks to these extensions it is possible to match on the SRH fields, this allow to have a full control on where the packets is directed (the next SIDs) and which nodes it has crossed before. At the kernel level, we implemented two additional kernel modules: the ip6t_srh as match extension and ip6t_SEG6 as target extension.

The ip6t_srh module implements the *SRv6 pre-processor*, the *Inner match*, and the *SRH match* from the advanced SERA architecture. The ip6t_SEG6 module implements the *Extended Action*. It is a new target (SEG6) for iptables rules that supports a set of SRv6-specific actions. To support the advanced mode SERA at user-space level, we extended the iptables user-space utility with two new shared libraries: libip6t_srh and libip6t_SEG6. They allow the iptables user to define SERA rules. These rules can have attributes from outer packet, inner packet, and SRH. Listing 4.4 shows a list of match options supported by the libip6t_srh extension. The ibip6t_SEG6 extension supports the new SR (SEG6) target with some SRv6-specific actions (shown in Listing 4.5).

For SRH programmed actions, we introduced a new sysctl variable (ip6t_seg6_args) that defines the number of rightmost bits in the active SID to be used as ARGS. The *SEG6* target decodes the ARGS bits to decide which action should be taken on the packet. If the decoded value does not correspond to any of the supported actions (e.g., ACCEPT, DROP, QUEUE, seg6-go-next, seg6-skip-next, seg6-go-last, etc.), SERA will send back an ICMP Parameter

Listing 4.4: Options of srh match extension

```
$ip6tables -m srh -h
 srh match options:
  [!]--inner-src      addr[/mask] Inner packet src
  [!]--inner-dst      addr[/mask] Inner packet dst
  [!]--srh-next-hdr   next-hdr    SRH Next Header
  [!]--srh-len-eq     hdr_len     SRH Hdr Ext Len
  [!]--srh-len-gt     hdr_len     SRH Hdr Ext Len
  [!]--srh-len-lt     hdr_len     SRH Hdr Ext Len
  [!]--srh-segs-eq    segs_left   SRH Segments Left
  [!]--srh-segs-gt    segs_left   SRH Segments Left
  [!]--srh-segs-lt    segs_left   SRH Segments Left
  [!]--srh-last-eq    last_entry  SRH Last Entry
  [!]--srh-last-gt    last_entry  SRH Last Entry
  [!]--srh-last-lt    last_entry  SRH Last Entry
  [!]--srh-tag        tag         SRH Tag
  [!]--srh-psid       addr[/mask] SRH previous SID
  [!]--srh-nsid       addr[/mask] SRH next SID
  [!]--srh-lsid       addr[/mask] SRH last SID
```

Listing 4.5: Options of SEG6 target extension

```
$ip6tables -j SEG6 -h
 SEG6 target options:
  [--seg6-action  action]
  Valid SEG6 actions:
  seg6-go-next        SEG6 go next
  seg6-skip-next      SEG6 skip next
  seg6-go-last        SEG6 go last
  seg6-eval-args      SEG6 eval args
```

Problem message point to the active SID. Such ICMP message can be used to understand which actions are supported by the firewall.

4.3.4 SERA Performance

Testbed description

In order to verify the correctness of SERA implementation and to evaluate the performance aspects, we designed a testbed environment that can be easily replicated, shown in Figure 4.4. For the experiments described in this section, we have deployed the testbed on CloudLab [72, 73]. Cloudlab is a flexible infrastructure dedicated to scientific research on the future of cloud computing.

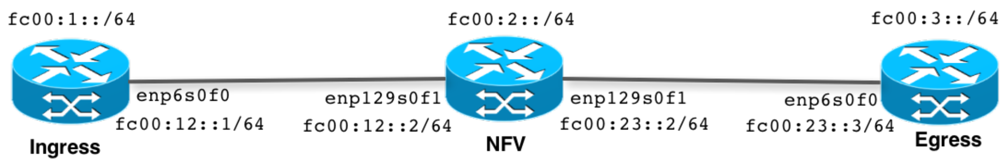


Figure 4.4: Performance evaluation testbed.

Our testbed is composed of three identical nodes. Each node is a bare metal server with Intel Xeon E5-2630 v3 processor with 16 cores clocked at 2.40GHz, 128 GB of RAM and two Intel 82599ES 10-Gigabit network interface cards. The three nodes are Linux servers and respectively represent an *ingress*, *NFV* and *egress* nodes of an SRv6 based SFC scenario. The *ingress* and *egress* nodes are running Linux kernel 4.14 [74] and have the 4.14 release of *iproute2* [75] installed. The NFV node runs a compiled Linux kernel 4.15-rc2 with SRv6 enabled and SERA firewall included [68].

The links between any two nodes X and Y are assigned IPv6 addresses in the form fc00:xy::x/64 and fc00:xy::y/64. For example, the two interfaces of the link between the ingress node (node 1) and the NFV node (node 2) are assigned the addresses fc00:12::1/64 and fc00:12::2/64. Each node owns an IPv6 prefix to be used for SRv6 local SID allocation. The prefix is in the form fc00:n::/64, where *n* represents the node number. For example, the NFV node (node 2) owns the IPv6 prefix fc00:2::/64. SRv6 local SIDs are in form LOC:FUNCT:ARGS, where LOC is the most significant 64-bits, ARGS is rightmost 16-bits and FUNCT is the 48-bits in between LOC and ARGS.

The *ingress* node is used as a source for SRv6 encapsulated traffic and the NFV node runs the SERA firewall inside a network namespace. The SERA firewall is instantiated on the SRv6 local SID fc00:2::f1:0/112. We have two destination servers $d1$ and $d2$ that are used as traffic sinks. Each destination server is assigned a prefix in the form fc00:dn::/64, where n is the destination server number. We configured the *ingress* node with two different SRv6 SFC policies as shown in Listing 4.6. The first SRv6 SFC policy is used to encapsulate traffic destined to *d1* as SRv6 packets in encap mode, while the second one encapsulates traffic destined to *d2* as SRv6 packets in insert mode. The SRv6 SFC policies are used to steer traffic through the SERA firewall, then to the egress node which removes SR encapsulation from packets as they leave the SR domain towards destinations (d1 and d2).

Listing 4.6: SRv6 SFC policy

```
# SR SFC policy - encap mode
$ ip -6 route add fc00:d1::/64 encap seg6 mode \
  encap segs fc00:2::f1:0,fc00:3::d6 dev enp6s0f0

# SR SFC policy - insert mode
$ ip -6 route add fc00:d2::/64 encap seg6 mode \
  inline segs fc00:2::f1:0,fc00:3::d6 dev enp6s0f0
```

We used *iperf* [76] to generate traffic on the ingress node. All traffic generated by *iperf* goes through the SRv6 SFC policies configured on the ingress node. In order to saturate the CPU of the NFV node, we used only one processor core for processing all the received packets by disabling the *irqbalance* service and assigning the *IRQ* for all interfaces to be served by the same CPU core.

Validation

In order to evaluate the performance of our implementation, we generated SRv6 traffic with a rate of 1 Mpps (10^6 packets per second). Each packet has a payload size of 1 KB. We wanted to measure the processing capacity (or throughput) of the firewall in processed packets per second (pps). We configured iptables with a rule that drops all traffic going from ingress node towards the destinations. Therefore, the counter of this rule represents the number of SR packets that the firewall has been able to process.

In order to evaluate the performance for different numbers of rules, we add a sequence of $N-1$ non-matching rules before the matching rule. In particular, we repeated each experiment for ten different numbers of rules *N* from 1 to 512. Each value plotted in Figures 4.5-4.9 represents the average of 30 runs, each run with a duration of 60 seconds. The confidence intervals are so close to the average that we have not plotted them.

We conducted five experiments as follows:

- Exp. 1: default iptables on plain IP packets.
- Exp. 2: basic mode SERA with SR encap mode.
- Exp. 3: basic mode SERA with SR insert mode.
- Exp. 4: advanced mode SERA with SR encap mode.
- Exp. 5: advanced mode SERA with SR insert mode.

In experiment 1 (default iptables), we used a rule that matches the IPv6 source and destination address of the received packets. The non-matching rules have the same structure, but different source and destination addresses. With only one rule configured (N=1), the throughput is 911 Kpps. As expected, the achieved throughput decreases with the number of rules, as shown in Figure 4.5. This is due to the operations that are executed for each rule. In particular, the function ip6_packet_match() is called for each rule.

In experiments 2 and 3, we evaluate the throughput of basic mode SERA with the same rules as the ones in the experiment 1 (matching the source and destination address). In these experiments, we are considering SR encapsulated packets and we set the ip6t_seg6 sysctl to apply the rule to the original packets. When there is only the matching rule ($N = 1$) the throughput is 875 Kpps in encap mode and 873 Kpps in insert mode (Figure 4.5). For larger *N*, the degradation of the performance is more evident.

The performance reduction of basic SERA with respect to iptables default is due to the *SRv6 pre-processor* module.This module has the task to look for the inner IPv6 header in the packet or for the SRH header in case of insert mode (we have re-used the ipv6_find_hdr function used by iptables). These operations are computationally expensive and are the reason for the reduction of the throughput visible in Figure 4.5. According to the design philosophy of iptables, the *SRv6 pre-processor* is executed once for each rule, because each rule operates in a stateless way and no state related to the packet is saved. From a performance point of view, this is clearly not efficient. Therefore, in order to improve the throughput result shown in Figure 4.5 we have considered an alternate design which achieves higher performance when a large number of rules may need to be applied to the packets and will be described at the end of this subsection. The insert mode has lower throughput than the encap mode due to our implementation of the *SRv6 pre-processor* block, which detects SRv6 packets in encap mode before those in insert mode. We decided to add the encap mode detection before the insert mode since it works also for *IPv6-in-IPv6* tunnels.

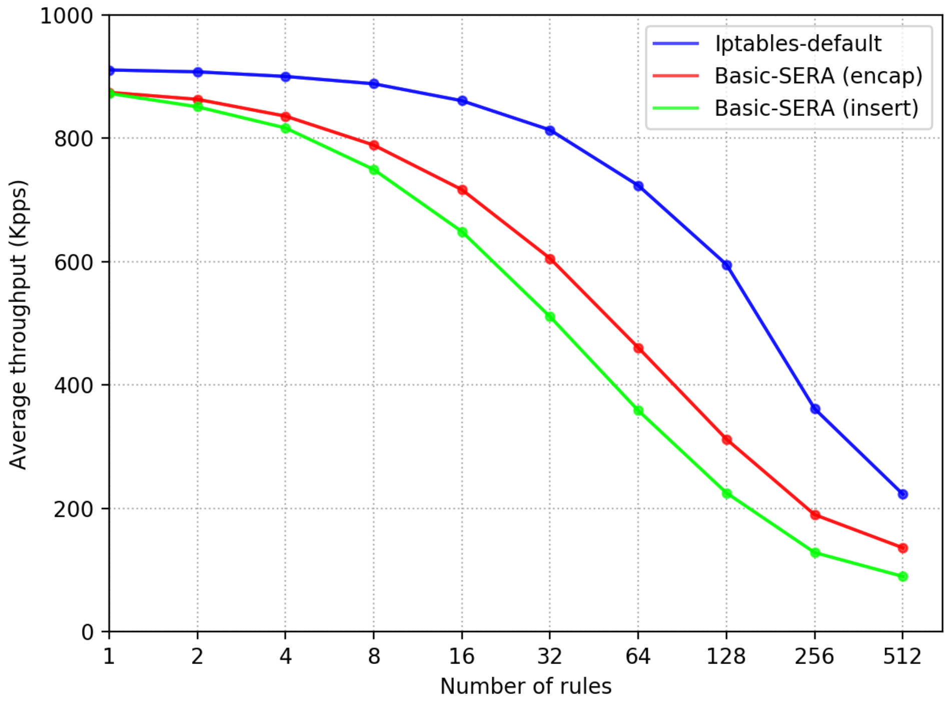


Figure 4.5: Basic SERA vs. default iptables

In experiments 4 and 5, we evaluated the throughput of advanced mode SERA. We considered an extended rule that matches source and destination address from both inner and outer packet. The results represented in Figure 4.6 are similar to the basic mode SERA, the throughput is 857 Kpps in encap mode and 849 Kpps in insert mode when one rule is configured ($N = 1$) and the performance degradation with respect to the default iptables is higher when the number of rules N increases. In Figure 4.7, we compare the throughput of basic and advanced mode SERA, considering the SR packets in encap mode. Both in the basic and in the advanced mode the *SRv6 pre-processor* is executed once for each rule, the advanced mode SERA achieves a lower throughput because it has to perform two match operations (Inner and Outer) rather than a single one.

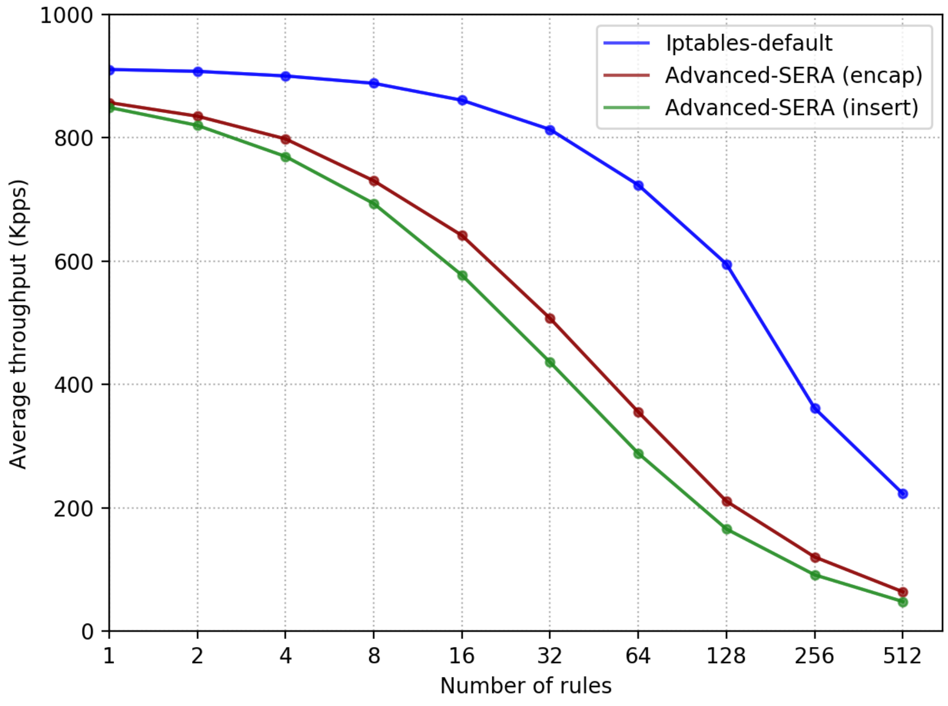


Figure 4.6: Advanced SERA vs. default iptables

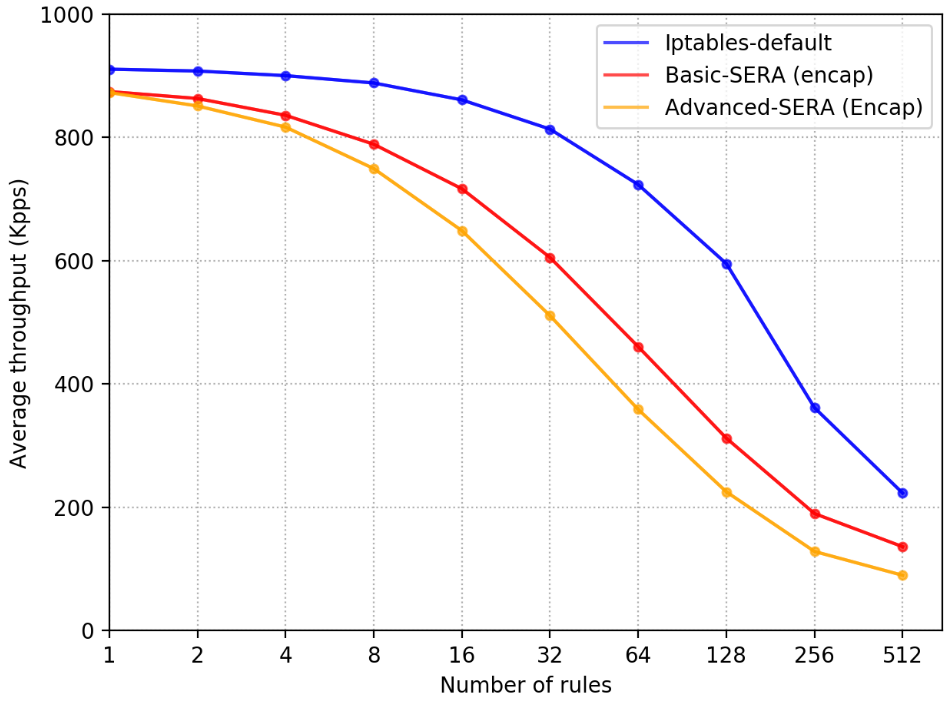


Figure 4.7: Basic SERA vs. Advanced SERA (encap mode)

The throughput reduction when several rules per packet are executed was not caused by problems in our implementation. To verify, we conducted a new experiment using an already existing iptables extension, the Routing Header extension. This extension is implemented in the ip6t_rt kernel module and able to match the common fields of the IPv6 Routing Header, including the Routing Type field. We run the test for the different numbers of rules *N* as in the previous experiments. For matching we used an extended rule that drops packets with Routing Type 4 (SRH). As shown in Figure 4.8 the obtained throughput perfectly matches our SERA implementation, confirming that the poor performance is inherently related to the iptables design.

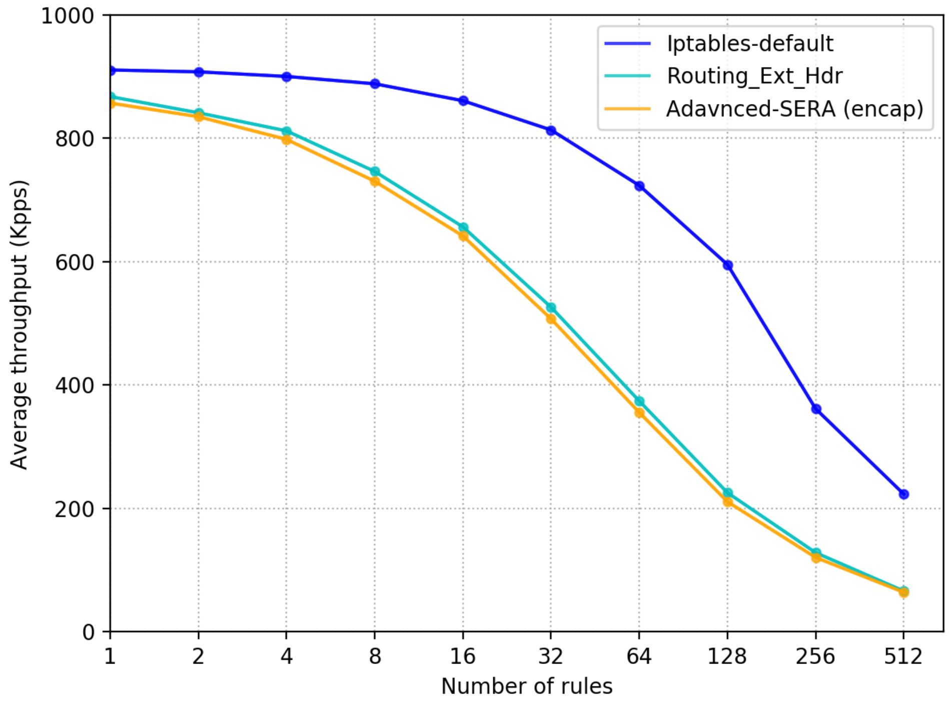


Figure 4.8: Existing RH iptables extension vs. advanced SERA

Finally, we tackled the issue of performance degradation and we were able to design and implement a solution focusing on one specific scenario, the basic mode SERA operating on SRv6 packets encapsulated in encap mode. In this scenario, a set of existing rules needs to be applied to the original packets that are encapsulated with IPv6-in-IPv6. As shown in Figure 4.5, there is a performance penalty which becomes significant when the number of rules is large. We revised the design of our iptables extension so that we can execute the SRv6 pre-processor once for each packet instead of re-executing it for every rule. The idea is to modify the pointers that point to the memory area in which the headers of the packet is stored once before executing all the rules and then to properly keep into account these modifications in the processing of the results of the matching. The throughput measurements of the revised design are shown in Figure 4.9. Only in case of a single rule, the throughput is slightly reduced due to the operations that are performed once for the packet. When the number of rules increases, there is no throughput degradation as for the basic mode SERA, and the performance approaches the one of the default iptables operating on plain (not encapsulated) IPv6 packets. This solution is integrated yet in the main line of the Linux. However, the open source implementation of the solution are available on GitHub to be re-used by the research community to use.

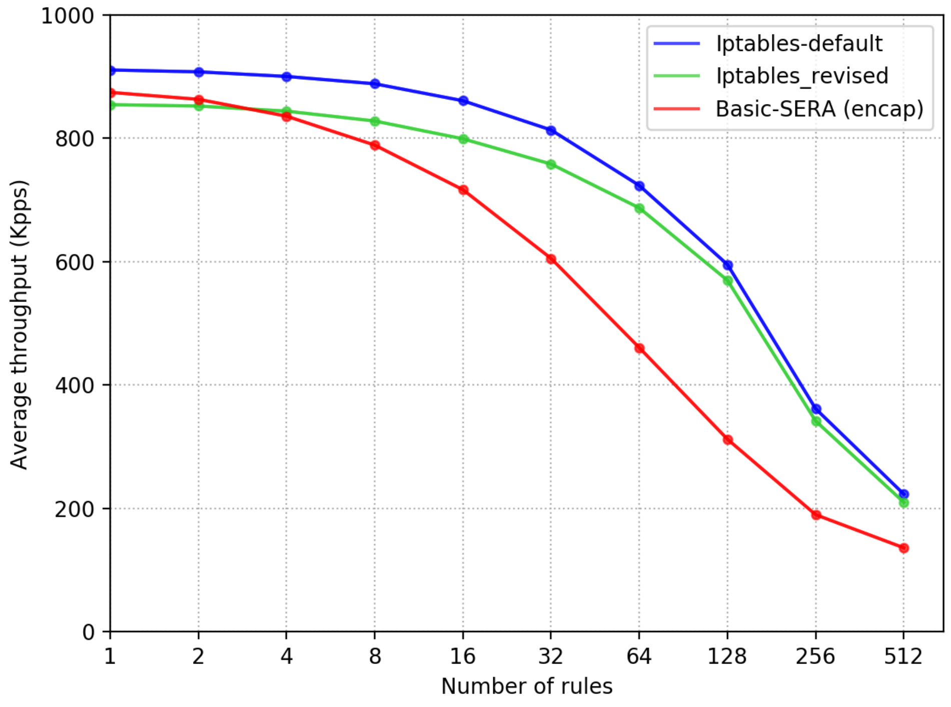


Figure 4.9: Revised iptables design vs. Basic SERA

4.4 SR-Snort

Snort is an open source, rule-based, network intrusion detection and prevention system [64] [77]. It combines the intelligence of several attack detection methods including signature-based, protocol-based, and anomaly-based to deliver flexible protection from malware attacks. Snort can also be used as a packet sniffer, similar to *tcpdump*[66], to read packets from a specific network interface and prints their headers fields. Snort supports three different run-time modes as follows:

- Sniffer mode: the packet analyzer mode of snort, similar to *tcpdump*[66], that simply reads packets off of the network, dissect them and print out the TCP/IP packet headers in a continuous stream on the console (screen).
- Packet Logger mode: logs the packets to disk for later analysis.
- Network Intrusion Detection System (NIDS) mode: Snort uses its detection engine to analyze the traffic and detect threats. In this mode Snort can be configured to act in either Passive (IDS) or Active (IPS) way. In the IDS mode snort gets a copy of each received packet, analyzes the packet and gives an alert if it detects a threat but without dropping the packets. In the IPS mode, Snort is inline in the packets data-path by receiving packets, analyzing them and decides if a packet should be forwarded further or drop it (if a threat is detected).

The design of Snort was initially relying on direct calls to the *libpcap* [78] library functions to acquire network packets. This design was changed in release 2.9 of Snort by introducing the data acquisition library (DAQ), which provides an abstraction layer between Snort and the different types of network interfaces [79]. DAQ offers a variety of modes for packets acquisition that can be chosen at run time. Currently, the DAQ module provides several options to acquire packets from the network such as *Pcap*, *AFPACKET*, *NFQ*, *IPQ*, *IPFW* and *Dump*.

Figure 4.10 shows the packet processing architecture of Snort. In a Linux environment, the DAQ module runs on top of the Linux network stack. First, Snort acquires packets from the DAQ module by using the *Acquire* module. Next, the *Decoder* module decodes packets and builds up the Snort's main data structure (SFSnortPacket), which contains the information required to process the packet. Then, the Snort *preprocessors* plugins (if any) are invoked. The *Preprocessors* provide a modular way to easily extend the functionalities of Snort by developing custom plugins. The *Log* module logs the packet information. After that, the *Detection* module compares the SFSnortPacket against the configured Snort rules. Finally, the *Verdict* module returns the action to be performed on the packet (e.g., accept or drop).

SR-Snort is an extended SR-aware version of Snort. It can apply Snort rules to inner packets of SRv6 encapsulated traffic. The packet processing architecture of SR-Snort is shown in Figure 4.11. We extended the packet processing architecture of Snort by adding two new

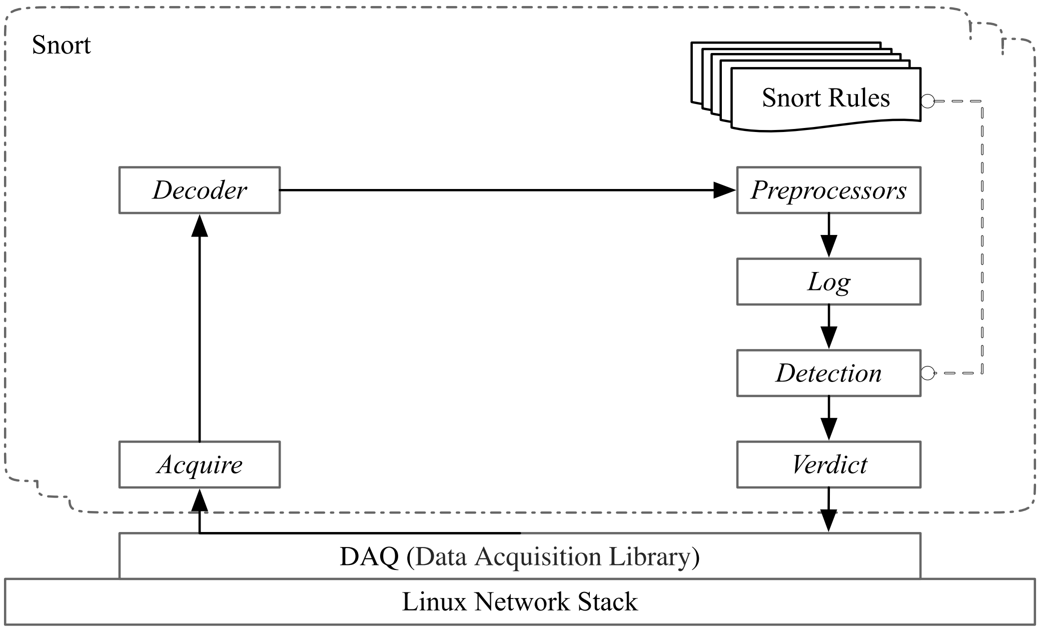


Figure 4.10: Snort packet processing architecture.

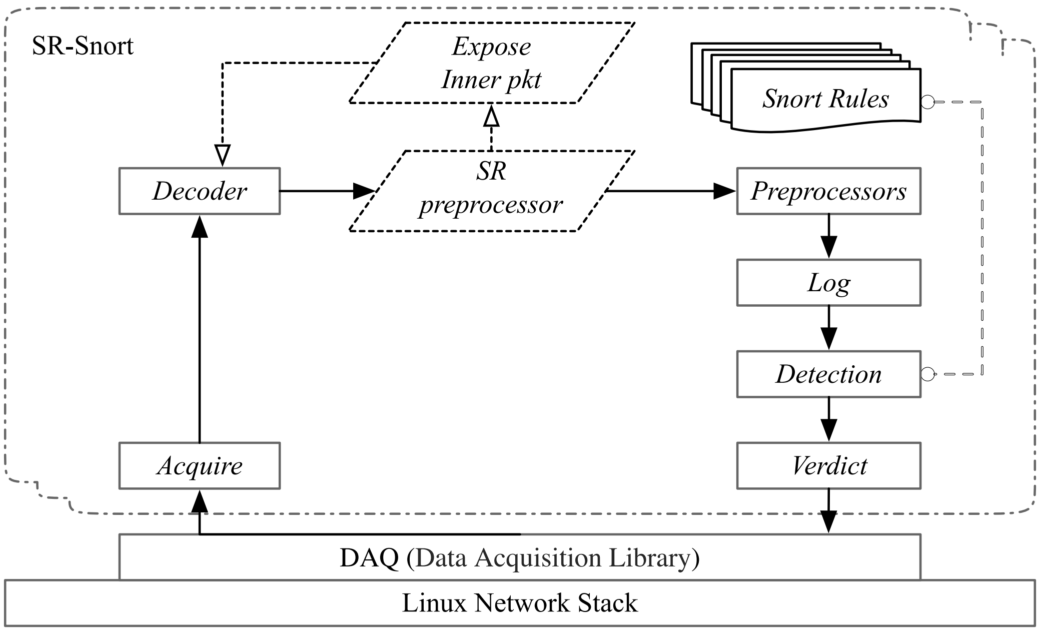


Figure 4.11: SRv6-Snort packet processing architecture.

modules: *SR preprocessor* and *Expose Inner pkt*. The *SR preprocessor* module is invoked immediately after the *Decoder* to detect SRv6 encapsulated packets, and hence classify packets into SRv6 and non SRv6. For non SRv6 packets, there is no change in the processing pipeline describe previously. SRv6 packets include SR encapsulation and to be able to apply Snort rules to the inner packet, the SRv6 encapsulation (i.e, Outer IPv6 Header and SRH) has to be removed before invoking the *Preprocessors*, the *Log*, and the *Detection* modules. SRv6 packets are directed to the *Expose Inner pkt* module, which removes the SRv6 encapsulation from packets and feeds them to the Snort *Decoder* to build up a new SFSnortPacket. The new SFSnortPacket has the information of the inner packet. Accordingly, Snort rules are applied to the inner packet. SR-Snort can be included in an SRv6 policy, where it uses the same set of legacy Snort rules and applies them to the exposed inner packet. SR-Snort supports inner IPv4 and IPv6 packets and can work in either IDS or IPS mode. We implemented SR-Snort by extending the open source implementation of Snort. We added support for the two new modules: *SR preprocessor* and *Expose Inner pkt*. The implementation is open source and publicly available on GitHub [80]. SR-Snort design, implementation and demo is published in [81].

4.5 SR-nftables

nftables is the next generation Linux firewall [65]. It has been merged in the release 3.11 of the Linux kernel and aims to replace the popular iptables firewall. *nftables* is built on top of the Linux netfilter framework and reuses many other existing kernel components such as the connection tracking system, NAT, userspace queueing and logging subsystem. It provides a new in-kernel packet classification framework that is based on a network-specific Virtual Machine (VM) and a new userspace command line tool, named *nft*.

The firewall rules in *nftables* are defined through the *nft* command line tool are compiled into bytecode in the userspace and then pushed this into the kernel via the *nftables* API. In case of firewall rules from the kernel, the bytecode is retrieved and from the kernel and then decompiled back to its original rules format in userspace. The main issues with *iptables* were rules scanning and writing/reading rules the firewall rules between userspace and the kernel. Rules in *iptables* are scanned sequentially which has a huge performance impact on packet throughput as the number of rules increases. Rules manipulation of a given table in *iptables* are atomic, which means adding/deleting/updating a rule will require reading the full table from the kernel to userspace, perform the required operation and write again the full table to the kernel. This model has been proved not to be scalable, hence the need for *nftables*.

nftables promises a higher performance and better scaling than *iptables* through the use of maps and concatenations to structure the ruleset to reduce the number of rules to be scanned. Another advantage of *nftables* with respect to *iptables* is that the intelligence is placed in userspace which means that adding a new feature does not require any change to the kernel in most of the cases.

We extended *nftables* with some SR-awareness features. We added new matching capabilities to allow matching traffic matching SRH information in *nftables*. The extension allows matching *Last Entry*, *Flags*, *Tag* and the *SID list* of SRH. In addition to these new matching features, we fixed some bugs related to IPv6 routing extension headers processing in *nftables*. We contributed our code to the netfilter project and it has been merged in release 0.8.4 of *nftables*[82].

4.6 SR-tcpdump

tcpdump is a widely used open source network protocols analyzer [66]. It used to print TCP/IP headers of packets on a given network interface card. Along with headers print out, it prints packets time stamps in units of hours, minutes, seconds, and fractions of a second. *tcpdump* provides several levels of details for printing packets as well as several filters to specify a subset of packets. In addition, it can be used to capture packets from a given interface and save them for later analyzing. The saved packets file are usually in the *pcap* format.

In the *tcpdump* implementation there is a dissector for each protocol to print out its header fields. One of the main issues we have faced while debugging SRv6 traffic is the lack of a tool to print out the SRH header fields which limited our visibility into the content of SRv6 packets. We implemented a new dissector that prints out the fields of the SRH. The SRH dissector has been merged in the mainline of the *tcpdump* implementation[83]. In addition we also fixed the checksum calculation for SRv6 traffic in *tcpdump* which has been merged in the its mainline [84].

Chapter 5

SRv6 performance

5.1 Introduction

SRv6 has been supported across several routers implementations, including open source software routers such as the Linux kernel and the Vector Packet Processing (VPP) [21], as well as hardware implementations from different network vendors [22]. SRv6 has several production deployment both in service providers networks and data centers [22]. Assessing non-functional properties of SRv6 such as scalability and fault tolerance, is therefore critical. In order to evaluate such properties, the availability of suitable performance evaluation tools is fundamental. In this work, we have addressed the design of a performance evaluation framework, which is a very challenging task [85]. In facts, packets are required to be forwarded at an extremely high rate using a limited CPU budget to process each of them. The IETF has defined the guidelines to evaluate the performance of forwarding devices [86]. The guidelines cover the test setup, packets formats and several measurements to be performed such as throughput, latency, jitter, and frame loss rate.

In this chapter, we present the design of SRPerf, an open source performance evaluation framework for SRv6, which we use to evaluate the performance of SRv6 behaviors of the Linux kernel, the Vector Packet Processor (VPP) [21] and the SREXT module [48]. The work described in this chapter has also been reflected in [87] [88]. The chapter is structured as follows. We discuss the principles of designing a performance evaluation framework in Section 5.2. We describe the design of our performance evaluation framework (SRPerf) in Section 5.3. In

Section 5.4, we describe the testbed used for the evaluation the performance of the SRv6 behaviors. In Section 5.5, we evaluate the performance of SRv6 behaviors in the Linux kernel. We compare the SRv6 performance in the Linux kernel and the Vector Packet Processor (VPP) in Section 5.6. Finally, we conclude the chapter with the state of the art of SRv6 performance in Section 5.7.

5.2 Designing a performance evaluation framework

The RFC 2544 [86] has defined a set of guidelines for network benchmarking including:

- **Testbed setup**. The benchmarking of a forwarding device requires a tester node with two NICs used as transmitting and receiving ports as shown in Figure 5.1. The transmitting and receiving ports of the tester node are connected respectively to the receiving and transmitting ports of the system under test (SUT) node. The tester node runs a traffic generator that sends packets to the SUT through the transmitting interface, and receives these packets after they have been forwarded by the SUT. Such setup allows the tester to assess the performance of the SUT by calculating the packet throughput, loss rate, and latency.
- **Device configurations.** The SUT must be configured to forward the received packets back to the tester node.
- **Frame size.** A performance test can use different frame sizes from the minimum frame size up to the maximum transfer unit (MTU). RFC2544 provides a list of frame sizes to be considered for testing, including 54, 64, 128, 256, 1024, 1518, 2048, and 4472 bytes.
- **Line rate**. The maximum number of frames per second that can be sent between the tester and the forwarding device is defined as Line rate, which is calculated by dividing the link maximum speed by the test frame size (including the transmission overhead). The transmission overhead is the number of bytes sent on the wire along with each frame for frame synchronization purposes between the sender and receiver. For example, considering Ethernet media, these bytes include CRC (4 bytes), preamble and SFD (8 bytes) and inter frame gap (12 bytes). The line rate is used as the maximum reference value to which one can relate the measured performance.

$$LineRate[pps] = Linkspeed/(framesize + transmission\ overhead)$$

- **Trial description.** Performance experiments require several repetitions of the same test. Each of these tests is called a trial, and should have a consistent and concrete description to make sure that the measurements are accurate across the several trials.
- **Trial duration.** The duration of each trial depends on the forwarding behavior being tested, and should be long enough to avoid abnormal results.
- **Benchmarking measures.** The IETF has defined several measures to assess the performance of forwarding devices including throughput, latency, jitter, and frame loss rate.

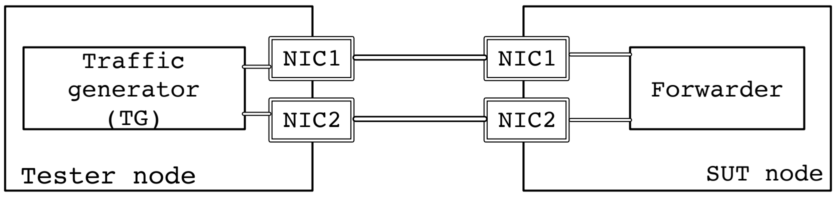


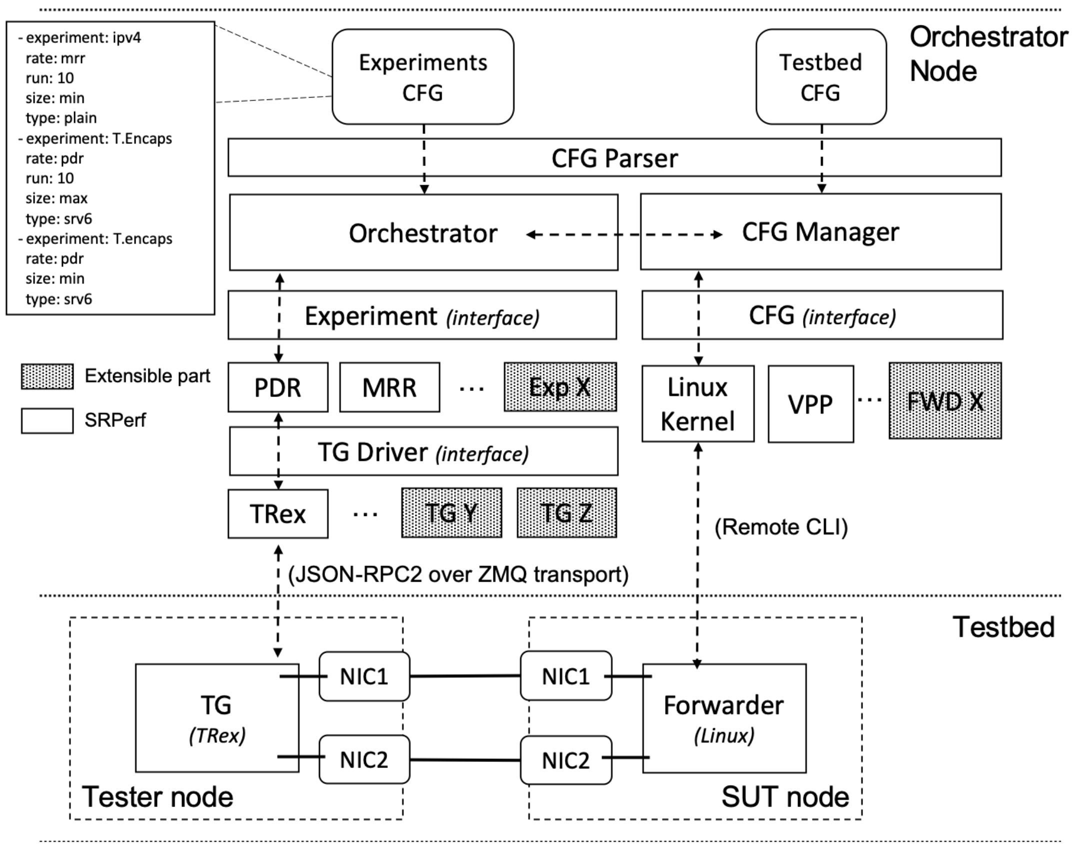
Figure 5.1: Testbed setup for performance evaluation experiments.

5.3 SRPerf

In this section, we illustrate our performance evaluation framework (SRPerf). In particular, we describe the internal design and the high-level architecture of SRPerf in Section 5.3.1. In Section 5.3.2, we elaborate on our evaluation methodology and describes the *Partial Drop Rate* (PDR) metric used to characterize the performance of a forwarding node. In Section 5.3.3, we explain the algorithm we developed for finding the PDR of a given forwarding behavior.

5.3.1 SRPerf Architecture

We designed SRPerf following the network benchmarking guidelines defined in RFC 2544 [86]. As shown in Figure 5.2, the architecture of SRPerf is composed of two main building blocks: the testbed and the *Orchestrator*. In turn, the testbed is composed by the Tester node and the System Under Test (SUT) node. These nodes have two network interfaces cards (NIC) each and are connected back-to-back using both NICs. The Tester sends traffic towards the SUT through one NIC, which is then received back through the other one, after being forwarded by the SUT. Accordingly, the Tester can easily perform all different kinds of throughput measurements as well as round-trip delay and jitter.



```
- experiment: ipv4
  rate: mrr
  run: 10
  size: min
  type: plain
- experiment: T.Encaps
  rate: pdr
  run: 10
  size: max
  type: srv6
- experiment: T.encaps
  rate: pdr
  size: min
  type: srv6
```

Figure 5.2: SRPerf architecture.

In our design, we rely on TRex [89] as traffic generator. TRex is an open source traffic generator based on the data plane development kit (DPDK) [47] for fast packet processing. TRex can generate layer-4 to layer-7 traffic at a rate up to 10-22 million packets per second (Mpps) using a single CPU core. It provides per-interface statistics, which are done at hardware level to provide higher precision. The statistics are used to calculate the various measures including throughput, latency, and jitter. Automation is another main feature of TRex, which is provided by the Python client API [90] and the scapy [91] packet manipulation program. As for the SUT Node, we currently support the Linux kernel and VPP as *Forwarder*.

We follow a top-down approach to describe the architecture of SRPerf. Two configurations files (upper part of the Figure 5.2) are provided as input to the *Orchestrator*. The first file, *Experiments CFG*, represents the necessary input to run the experiments. In particular, it defines: i) the type of experiment (i.e. set of SRv6 behaviors to be tested, type of tests and type of algorithm); ii) the number of runs; iii) the size and type of the packets to be sent between the traffic generator and the *Forwarder*. The second configuration file (*Testbed CFG*) defines the forwarding engine of the SUT and the information needed to establish a SSH connection with it. The SRPerf configuration files use a YAML [92] syntax, an example of configuration is reported in the upper-left part of the Figure 5.2.

The *Orchestrator* leverages the *CFG Parser* to extract the configuration parameters and to initialize the experiment variables. The *CFG Parser* is a Python module which uses *PyYAML* parser [93] to return Python objects to the caller. The *Orchestrator* is responsible for the automation of the whole evaluation process. According to the input parameters, it creates an *Experiment*; specifically, the *Orchestrator* uses different algorithms for calculating the throughput. Each algorithm offers an API interface (see Figure 5.2) through which the *Orchestrator* can run an *Experiment*. An example of currently supported throughput measurement algorithms is the Partial Drop Rate (PDR), described in Section 5.3.2. Moreover, the *Orchestrator* provides a mapping between the forwarding behaviors to be tested and the type of traffic required to test each behavior. For example, to test the *End* behavior, it is necessary to use an SRv6 packet with an SRH containing a SID list of at least two SIDs and the active SID must not be the last SID - the type of packet to be replayed during the experiments has to be passed to the experiment. The *Orchestrator* controls the TG (deployed in the Tester node) through the high level abstraction provided by the *TG Driver*, which translates the calls coming from the other modules in commands to be executed on the TG. Each driver is a Python wrapper that can speak native Python APIs or use any other transport mechanism supported by the language. For example, the TRex driver includes the Python client of the TRex automation API [90] that uses as transport mechanism JSON-RPC2 [94] over ZMQ [95]. The *Orchestrator* can be deployed on the same node of the TG or in a remote node.

The *CFG Manager* controls the forwarding engine in the SUT. It is responsible for enforcing the required configuration in the *Forwarder*. The *Orchestrator* provides the mapping between the forwarding behaviors to be tested and the required configuration of a given forwarding engine. Hence, the *Orchestrator* is able to properly instruct the *CFG Manager*. For each forwarding engine, we implement a *CFG* which provides the *CFG Manager* with the means to enforce a required configuration. In particular, a *CFG* is a bash script defining a configuration procedure for each behavior to be tested. The configuration is applied via the Command Line Interface (CLI) exposed by the forwarder. For example, to test the *End* behavior in the Linux kernel, we implement a bash procedure called *end*. In this procedure, we leverage the *iproute* utility to configure the forwarding engine in the SUT with two FIB entries: 1) an SRv6 SID with the *End* behavior; 2) a plain IPv6 FIB entry to forward the packet once the *End* function has been performed. The configuration can be as simple as adding a FIB entry to forward the received packets back to the Tester, but also being a more complex configuration that manipulates the incoming packets before forwarding them back to the Tester. The *CFG Manager* first pushes the *CFG* scripts in the SUT and then applies a given configuration running commands over an SSH connection.

The SRPerf implementation is open source and available at [96]. SRPerf is mostly written in Python, and provides a toolset to facilitate the deployment of experiments. It offers an interface for the automatic generation of configuration files. Moreover, it provides different configuration scripts to deploy experiments on any commodity hardware. These scripts include installation and initial configuration of both the *TG* and the *Forwarder*. The framework is modular and can be expanded in different ways. It can be extended to support new traffic generators by simply creating a new driver for each. A new forwarding behavior can be added by updating the *CFG Manager* with the configuration required for such behavior. New algorithms for calculating throughput and delay can be developed and plugged into the *Orchestrator*. It can support different *Forwarders* in the SUT, which only requires the *CFG manager* to be updated to recognize them and to implement the related *CFG* object. In this thesis, we have first considered the Linux kernel networking as Forwarder and then, leveraging the framework described above, we added the support for VPP software router.

5.3.2 Evaluation methodology

RFC 1242 [97] defines the *Throughput* as the maximum rate at which all received packets are forwarded by the device. It is a standard measure to compare performance of different network devices. The throughput can be measured in bits per second (bps) as well as packets per second (pps). The FD.io CSIT report [98] defines *No-Drop Rate* (NDR) and *Partial Drop Rate* (PDR). NDR is the highest throughput achieved without dropping packets, so it corresponds to the *Throughput* defined by RFC 1242. PDR is the highest throughput achieved without dropping traffic more than a pre-defined loss ratio threshold [99]. We use the notation PDR@X%, where X represents the loss ratio threshold. For example, we can evaluate PDR@0.1%, PDR@0.5%, PDR@1%. NDR can be described as PDR@0%, i.e. PDR with a loss threshold of 0%. Considering that *throughput* can be used with wider meanings, the terminology defined in [98] (e.g. No-Drop Rate) is clearer and it will be used hereafter. Hence, we can use throughput to refer in general to the output forwarding rate of a device. In this thesis, we will consider only the Partial Drop Rate (PDR) since it is more generic than the NDR.

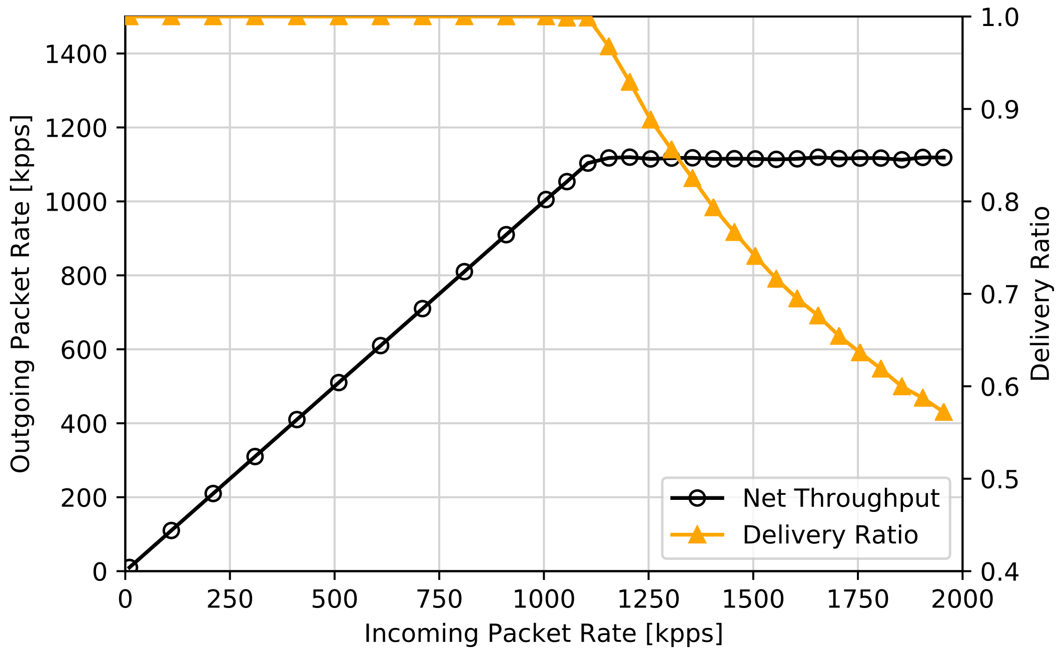


Figure 5.3: Throughput vs Delivery Ratio

Finding the PDR requires scanning of a broad range of possible traffic rates. In order to explain the process, let us consider the plain IPv6 forwarding in the Linux kernel. Figure 5.3 plots the throughput (i.e. the output forwarding rate) and the Delivery Ratio (DR) versus the input rate, defined and evaluated as follows. We generate traffic at a given PS packet rate [kpps] for a duration D [s] (usually $D = 10s$ in our experiments). Let the number of packets generated by the TGR node and incoming to the SUT in an interval of duration D be P_{IN} (Packets INcoming in the SUT). We define the number of packets transmitted by the SUT (and received by the TG) as P_{OUT} (Packets OUTgoing from the SUT). The throughput T is P_{OUT}/D [kpps]. We define the Delivery Ratio as $P_{OUT}/P_{IN} = P_{OUT}/(PS * D) = T/PS$. For each incoming packet rate we run a number of test repetition (e.g. 15) to evaluate the average and standard deviation of the outgoing rate. Hence, the Delivery Ration is the ratio between the input and the output packet rates of a device, for a given forwarding behavior under analysis. It is 100% for all incoming data rates less than the device No-Drop Rate. Initially, the throughput increases linearly with the increase in the incoming rate. This region is often referred to as *no drop region*, i.e. where the Delivery Ratio is always 100%. If the forwarding process is CPU-limited, the CPU usage at the SUT node increases with the increase of incoming traffic rate (i.e. the sending rate of the Tester).

Ideally, the SUT node should be able to forward all received packets until it becomes 100% CPU saturated. On the other hand, in our experiments with the Linux based SUT we measured very small but not negligible packet loss ratio in a region where we have an (almost) linear increase of the Throughput. Therefore, it is better to consider the Partial Drop Rate (PDR) and we used 0.5% as threshold. The PDR@0.5% is the highest incoming rate at which the Delivery Ratio is at least 0.995. The usefulness of the PDR is that it allows to characterize a given configuration of the SUT with a single scalar value, instead of considering the full relation between Throughput and Incoming rate shown in Figure 5.3. The procedure for finding the PDR for a given loss threshold is described in the next section.

5.3.3 PDR finder algorithm

Estimating the PDR of a given forwarding behavior is a time consuming process, since it requires the scanning of a broad range of possible traffic rates. In order to automate the PDR finding process, we have designed and developed the PDR finder algorithm. It scans a range of traffic rates with the objective of estimating the PDR value. Alg. 1 reports the pseudo code of

Algorithm 1 PDR finder algorithm

```
1: $minRate \leftarrow startingTxRate$
2: $maxRate \leftarrow lineRate$
3: loop
4:   // The algorithm terminates when the size of the searching window is less than a threshold
5:   if $|maxRate - minRate| \leq \varepsilon$ then
6:     return $[minRate, maxRate]$
7:   end if
8:   // Evaluate the DR for the window middle point
9:   $txRate \leftarrow \frac{minRate + maxRate}{2}$
10:  $rxRate \leftarrow runExperiment(txRate)$
11:  $deliveryRatio \leftarrow \frac{rxRate}{txRate}$
12:  // Halve the size of the searching window
13:  if $deliveryRatio < pdrThreshold$ then
14:    $maxRate \leftarrow txRate$
15:  else
16:    $minRate \leftarrow txRate$
17:  end if
18: end loop
```

the PDR finder algorithm. It performs a logarithmic search in the space of possible solutions which is upper limited by the line rate of the NICs (see lines 1 and 2). It returns an interval $[a,b]$ of traffic rates estimating with a given confidence (ε) the PDR value. The maximum interval distance ε is a configurable option that tunes the algorithm precision. The algorithm starts to decrease the amplitude of the searching window until such value becomes less than the minimum interval width ε (line 5). At each iteration (loop starting at line 3), the size of the searching window is halved and the Delivery Ratio (DR) is evaluated for the window middle point, which is considered to be the current traffic rate (from line 13 to line 17). If the Delivery Ratio of the middle point is less than the threshold, the upper bound of the window is set to the current rate. Otherwise, the lower bound of the searching window is set with the current rate. This process is iterated until the exit condition is triggered: the algorithm terminates when the difference between a and b is less or equal than ε ($b - a \leq \varepsilon$).

5.4 Testbed description

We deployed on CloudLab [72] our testbed as illustrated in Figure 5.2. CloudLab is a cloud infrastructure dedicated to scientific research on the future of cloud computing. Our testbed is composed of two identical nodes (Tester and SUT). Each of these nodes is a bare metal server equipped with 16 cores Intel Xeon E5-2630 v3 processor clocked at 2.40GHz and 128 GB of RAM. Each server has two Intel 82599ES 10-Gigabit network interface cards. The tester is running TRex [89] as a traffic generator and has the TRex Python automation libraries installed [90]. We used the testbed to evaluate the performance of the SRv6 behaviors both in the Linux kernel (Section 5.5) and VPP (Section 5.6). The reported results are measured using a single CPU core. The single CPU tests are used to assess the base forwarding performance on commodity servers.

or the Linux kernel experiments, the SUT machine is running a vanilla Linux kernel (release 5.2) and has the *iproute2* [75] tool (release 5.2) installed. The *ethtool* (release 5.2) is installed to configure the NIC hardware features such as hardware offloading [100]. We disabled the CPU hyper-threading feature of the SUT to measure the performance of single physical core, since the hyper-threading impacts the forwarding performance of servers [101] [102]. We tuned the testbed to force all received traffic to be processed by a single (physical) CPU core. We rely on the receive-side scaling (RSS) [103] and SMP IRQ affinity [104] features. RSS is responsible for distributing the received packets across several hardware-based receive queues. The number of receive queues scales with the number of CPU cores. Each receive queue is assigned a CPU core to process its packets. The distribution of packets across the receive queues is based on a hash function which assigns packets of the same traffic flow to the same receive queue, hence being processed by the same CPU core. However, we also used the SMP IRQ affinity feature to assign all the receive queues to the same CPU core to guarantee the single CPU processing independently from the hash function feature. We disabled all hardware offloading capabilities of the NICs to measure the kernel performance independently of the NIC hardware features. We used the *ethtool* tool to disable the various NIC offloading features such as large receive offload (LRO) [105], generic receive offload (GRO) [106], generic segmentation offload (GSO) [107] and checksum offloading. For the VPP experiments, the SUT machine is running VPP release 19.04. We customized the VPP startup configuration to use single CPU core and disable all hardware offloading capabilities.

5.5 Performance evaluation of SRv6 in the Linux kernel

In this section, we focus on the SRv6 performance in the Linux kernel. The VPP performance is covered in Section 5.6. To evaluate the SRv6 performance in the Linux kernel, we performed a set of experiments that compare the performance of the SRv6 behaviors to the plain IPv6 forwarding. In these experiments we measured the PDR values for the various SRv6 behaviors. The PDR values (Kpps) and the ratio to the plain IPv6 forwarding (R_{IPv6}) of the experiments are reported in Table 5.1. The reported PDR values represent the mean value of 10 trials. Each of these trials has a duration of 10 seconds. The measurements reported in this section are available on GitHub [108]. We have classified these experiments in five categories as follows:

- Exp1: SRv6 transit behaviors.
- Exp2: SRv6 endpoint behaviors with no decapsulation.
- Exp3: SRv6 endpoint behaviors with decapsulation and lookup in a specific routing table.
- Exp4: SRv6 endpoint behaviors with decapsulation and cross-connect through an outgoing interface.
- Exp5: SRv6 proxy behaviors.

Experiment	Behavior	PDR/R_{IPv6}	IPv6
Exp1	T.Insert	1039.29	1221.05
		85.11%	
	T.Encaps	978.13	
		80.1%	
	T.Encaps.L2	828.89	
		67.88%	
Exp2	End	900.52	
		73.75%	
	End.T	979.25	
		80.19%	
	End.X	123.13	
		10.08%	
Exp3	End.DT4	1027.00	
		84.11%	
	End.DT6	960.06	
		78.63%	
Exp4	End.DX2	1299.15	
		100.66%	
	End.DX4	929.02	
		76.08%	
	End.DX6	122.76	
		10.05%	
Exp5	End.AD	640.88	
		52.49%	
	End.AM	759.57	
		62.21%	
Exp6	End.X (patched)	1228.71	
		100.63%	
	End.DX4 (patched)	1248.29	
		102.23%	
	End.DX6 (patched)	1213.06	
		99.35%	

Table 5.1: Performance for SRv6 behaviors in Linux (Kpps).

Exp1: SRv6 transit behaviors

In the Exp1 experiment category, we evaluated the performance of the three different SRv6 transit behaviors: *T.Insert*, *T.Encaps*, and *T.Encaps.L2*. The details of SRv6 transit behaviors processing are explained in Section 2.4.1. The results of the experiment are reported in the Exp1 section of Table 5.1 and shown in Figure 5.4.

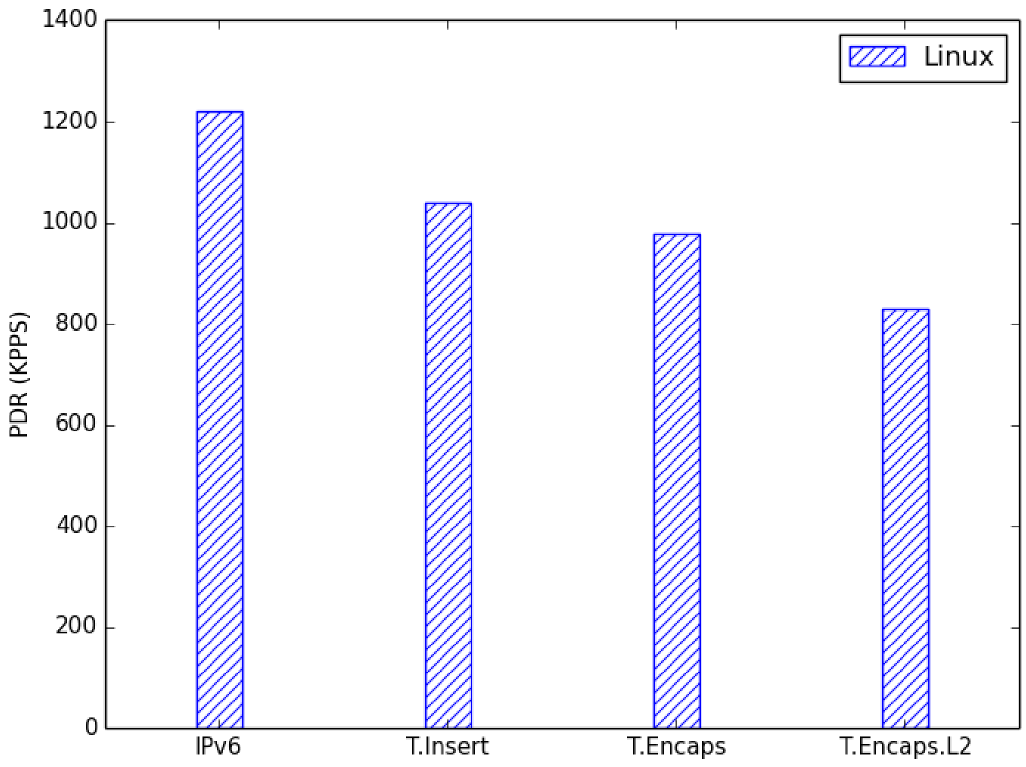


Figure 5.4: Performance measurements for Exp1.

The plain IPv6 in the Linux kernel has forwarding of ≈1221 kpps. To measure the plain IPv6 forwarding we used the main routing table of the Linux kernel and we have only one routing entry which forwards the packets to the TG in the *Tester node*. This represents the base-line forwarding of the IPv6 stack in the Linux kernel. Some performance measurements experiments might considers using routing tables of different sizes or use a routing table different from the main one. These kind of measurements is outside the scope of this thesis. The *T.Insert* shows a forwarding performance of ≈1039 kpps which is 85.11% of the performance of plain IPv6 forwarding. To understand the reasons for the 14.89% decrease of the forwarding capability, we recall that the *T.Insert* processing requires inserting an SRH between the IPv6 and transport headers of the IPv6 packet. Such insertion operation is done in the Linux kernel as follows: i) extending the head of packet buffer with the space required for SRH insertion; ii) move (i.e. copy) the IPv6 header bytes to the beginning of the buffer head to create a space between the IPv6 header and the transport header; iii) write the SRH in the created space between the IPv6 header and the transport header; iv) copy the first SID in the SID list of the inserted SRH into the destination address of the IPv6 header; v) forward the packet based on the updated destination address. These extra five operations with respect to plain IPv6 forwarding are the reason for the introduced degradation of forwarding performance. The SRv6 *T.Encaps* has a forwarding performance of $\approx$978 kpps which is $\approx$5% less than the SRv6 *T.Insert*. The reason for such 5% is that the *T.Encaps* behavior is required to add to the packet an outer IPv6 header in addition to the SRH. For the *T.Encaps.L2* behavior, the SUT node is able to forward $\approx$828.89 kpps. The performance of the *T.Encaps.L2* behavior is $\approx$12% less than the *T.Encaps*. We have analysed the motivations for this performance degradation. The root cause is that the current Linux implementation of *T.Encaps.L2* is not a real L2 encapsulation solution. It is not taking a L2 packet and encapsulating it inside an SRv6 packet. Rather, it is taking a L3 (IPv6) packet and re-constructing the L2 MAC header before encapsulating the packet with an IPv6 header and SRH. The re-construction of the L2 MAC header is performed at L3 because the L2 information are lost before the packet is processed at L3 by the SRv6 components. For these reason, the performance degradation of *T.Encaps.L2* is much higher than the other behaviors considered in this experiment. This analysis of the performance of the SRv6 behaviors in the Linux kernel shows that the current Linux implementation of the SRv6 *T.Encaps.L2* behavior does not follow the pseudocode defined in [34].

Exp2: SRv6 endpoint behaviors with no decapsulation

In the Exp2 experiment category, we evaluated the performance of three SRv6 endpoints behaviors: *End*, *End.X*, and *End.X*. These behaviors require the active SID not to be the last SID in the SRH SID list and do not remove the SRv6 encapsulation (IPv6 header and SRH) from packets (the reason for calling them "no decapsulation"). These SRv6 behaviors update the destination address of the received SRv6 packets to the next IPv6 address in the SID list. After the packet destination address is updated, it is sent to the next hop by doing route lookup in the main routing table (*End*), by doing route lookup in a specific routing table (*End.T*) or cross-connecting the packet through an outgoing interface towards the destination (*End.X*). The results of the experiment are reported in the Exp2 section of Table 5.1. Figure 5.5 shows the performance of the SRv6 endpoint behaviors with no decapsulation along with the plain IPv6 forwarding.

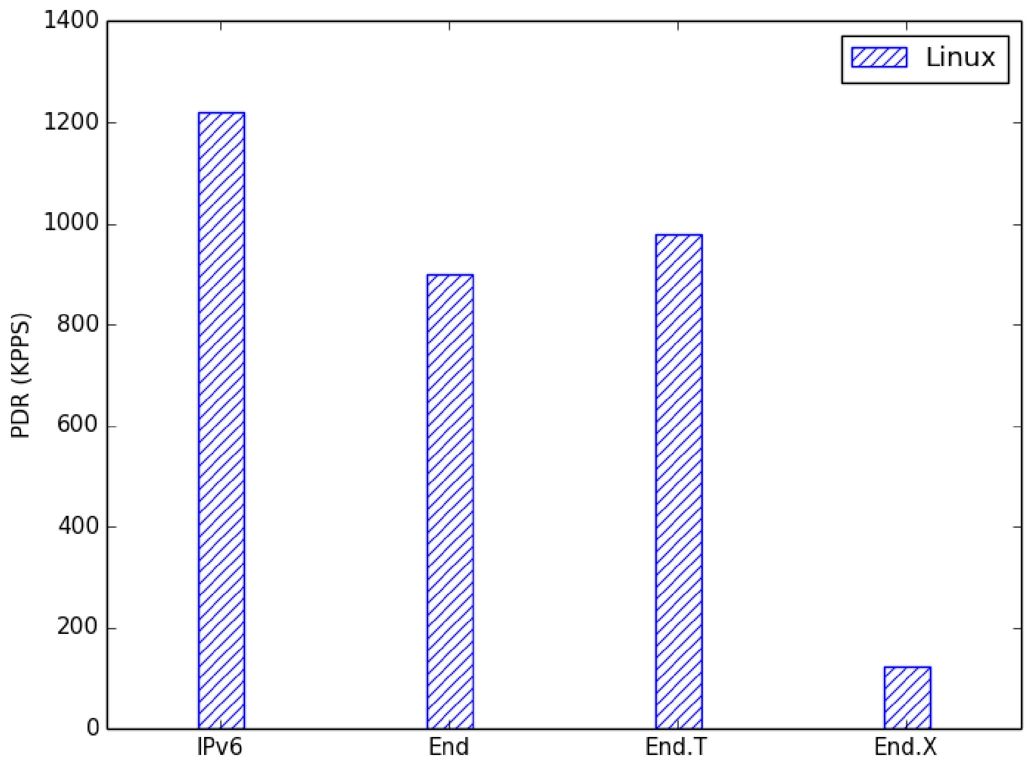


Figure 5.5: Performance measurements for Exp2.

In case of SRv6 *End* behavior, the *SUT* node is able to forward ≈900 kpps which is 73.75% of performance of the plain IPv6 forwarding. The reason for the 26.25% decrease of performance is due the extra processing that SRv6 *End* behavior performs with respect to plain IPv6 forwarding. These processing operations include: i) parse the SRH to get the next active SID, which is done using ipv6_find_hdr kernel function; ii) update the destination address of the packet to be the next active SID; iii) perform IP rules lookup to apply the policy-based routing (if any, not in our measurement experiment) and get the routing table to be used for routing lookup; iv) perform the IPv6 lookup using the packet destination into the the main routing table to find the next hop where the packet has to be sent. The SRv6 *End.T* behavior has a forwarding throughput ≈979 kpps, which is ≈7% higher than SRv6 *End* behavior. The SRv6 *End.T* behavior performs better than the SRv6 *End* since the routing table used for the lookup is defined by the control plane, hence the kernel saves the cost of performing IP rules lookup that are executed in case of the *End* behavior. The SRv6 *End.X* behavior shows a very poor performance of ≈123 kpps. It corresponds to a 90% drop of the performance compared to plain IPv6 forwarding. The analysis of the root causes of such poor performance is analysed and fixed in Section 5.5.1.

Exp3: SRv6 endpoint behaviors with decapsulation and lookup in a specific routing table

In the Exp3 experiment category, we evaluated the performance of two SRv6 endpoints behaviors: *End.DT4* and *End.DT6* which remove the SRv6 encapsulation from packets and forward the inner packet towards the next-hop by doing routing lookup in a specific routing table specified by the control plane. The results of the experiment are reported in the Exp3 section of Table 5.1. Figure 5.6 shows the performance of the SRv6 endpoint behaviors measured in Exp3 along with the plain IPv6 forwarding.

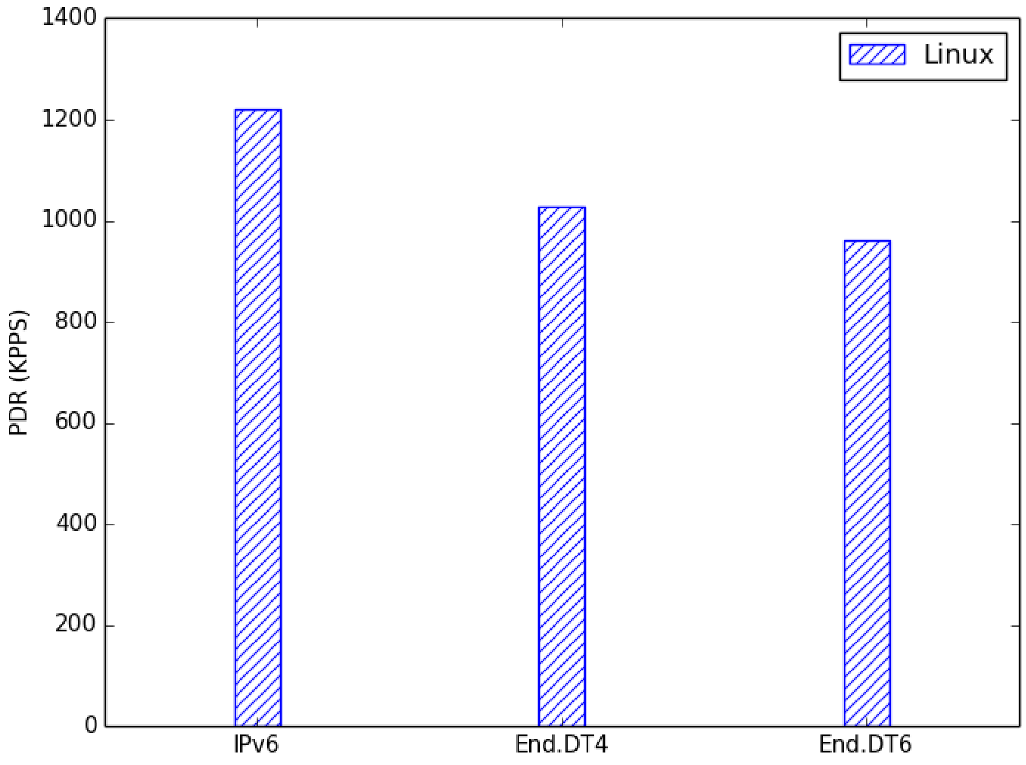


Figure 5.6: Performance measurements for Exp3.

The SRv6 *End.DT4* has a forwarding throughput ≈1027 kpps, which is ≈84.11% of the plain IPv6 forwarding. The reduction of the forwarding rate is due to the processing operations of SRv6 *End.DT4* that include: i) parse the SRv6 packet to determine the size the SRv6 encapsulation to be removed from the packet; ii) remove the SRv6 encapsulation from the packet buffer head; iii) forward the inner packet to the next hop towards the destination by doing IPv4 routing lookup in the routing table associated the SRv6 behavior. For the forwarding of the inner IPv4 packet, we used a secondary IPv4 routing table that has one a single routing entry. The SRv6 *End.DT6* has a forwarding throughput of ≈960 kpps, which is around 5.48%

less than the *End.DT4*. The reason for this additional decrease of the throughput is the routing lookup operation performed after removing the SRv6 encapsulation. SRv6 *End.DT6* does IPv6 routing lookup while *End.DT4* does IPv4 routing lookup. The SRv6 *End.DT6* behavior uses a secondary IPv6 routing table that has one a single routing entry to perform lookup for the inner IPv6 packet. In the Linux kernel, the IPv4 stack has a higher throughput than the IPv6 stack as witnessed by the results in Section 5.6.2.

Exp4: SRv6 endpoint behaviors with decapsulation and cross-connect through an outgoing interface

In the Exp4 experiment category, we evaluated the performance of three SRv6 endpoints behaviors: *End.DX2*, *End.DX4* and *End.DX6* which remove the SRv6 encapsulation from packets and forward the inner packet to the next-hop towards the destination by doing cross-connect through an outgoing interface to the next hop. The results of the experiment are reported in the Exp4 section of Table 5.1. Figure 5.7 shows the performance of the SRv6 endpoint behaviors measured in Exp4 along with the plain IPv6 forwarding.

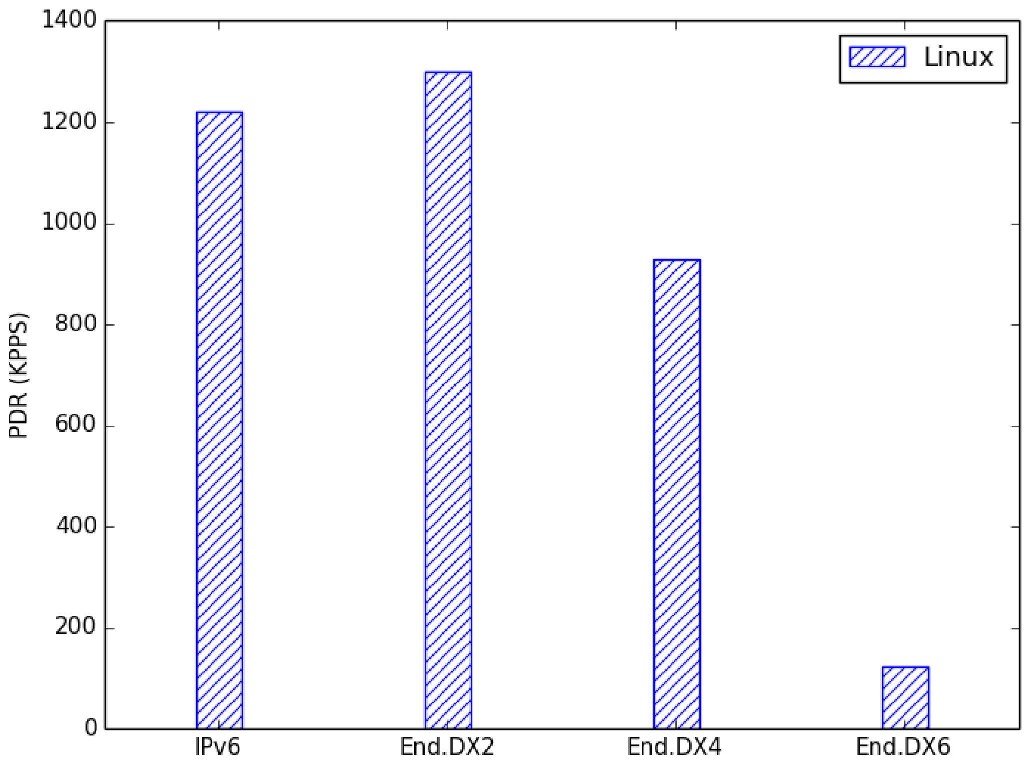


Figure 5.7: Performance measurements for Exp4.

The SRv6 *End.DX2* has a forwarding throughput of ≈1229 kpps which is ≈0.66% higher than the plain IPv6 forwarding. The reason why SRv6 *End.DX2* is performing better than plain IPv6 is that the kernel does not need to perform Layer-3 lookup once the packet has been decapsulated. Instead, it pushes the packet directly into the transmit queue of the interface towards the next-hop. SRv6 *End.DX4* has a forwarding throughput ≈929 kpps which is around 76% of the plain IPv6 forwarding. This result is unexpected, as *End.DX4* pushes the packet after being decapsulated directly into the transmit queue of the interface towards the next-hop. We expect to have a performance similar to *End.DX2*. The *End.DX6* shows a very poor performance of ≈122 kpps with around 90% drop in the performance compared to plain IPv6. The issues of both *End.DX4* and *End.DX6* behaviors are analysed and fixed in Section 5.5.1.

Exp5: SRv6 proxy behaviors (implemented in the SREXT module).

In the Exp5 experiment category, we evaluated the performance of two SRv6 proxy behaviors: *End.AD* and *End.AM*, which we have implemented in a Linux kernel module called SREXT. *End.AD* and *End.AM* are used to process SRv6 encapsulation on behalf of SRv6-unaware VNFs as explained in Section 3.5. The results of the experiment are reported in the Exp5 section of Table 5.1. Figure 5.8 shows the performance of the SRv6 endpoint behaviors measured in Exp5 along with the plain IPv6 forwarding. The *End.AD* and *End.AM* are implemented in SREXT [48] as explained in Section 3.6. In case of *End.AD*, SREXT is able to forward ≈640 kpps, which is around 52% of the forwarding throughput of plain IPv6. The 48% performance decrease is due to the CPU intensive operations that the proxy has to perform including: parsing SRv6 packets, remove the SRv6 information and save them to the cache, forward the packet to VNF, intercepts all packets processed by the VNF, retrieve the SR information from the cache and apply them to the packets coming from the VNF.

We believe that the achieved 640 kpps throughput is still acceptable as forwarding rate for VNFs running on commodity hardware. Considering the default IPv6 packet size (1280 bytes), *End.AD* can deliver ≈6.5 ($1280 * 8 * 640$) Gbps to SRv6-unaware VNFs. The *End.AM* behavior has a forwarding throughput of ≈759 kpps, which is around 10% higher than *End.AD*. The reason is that the *End.AM* does not remove the SRv6 information from the packet, instead it only updates the destination address to be the last SID in the SRH SID list.

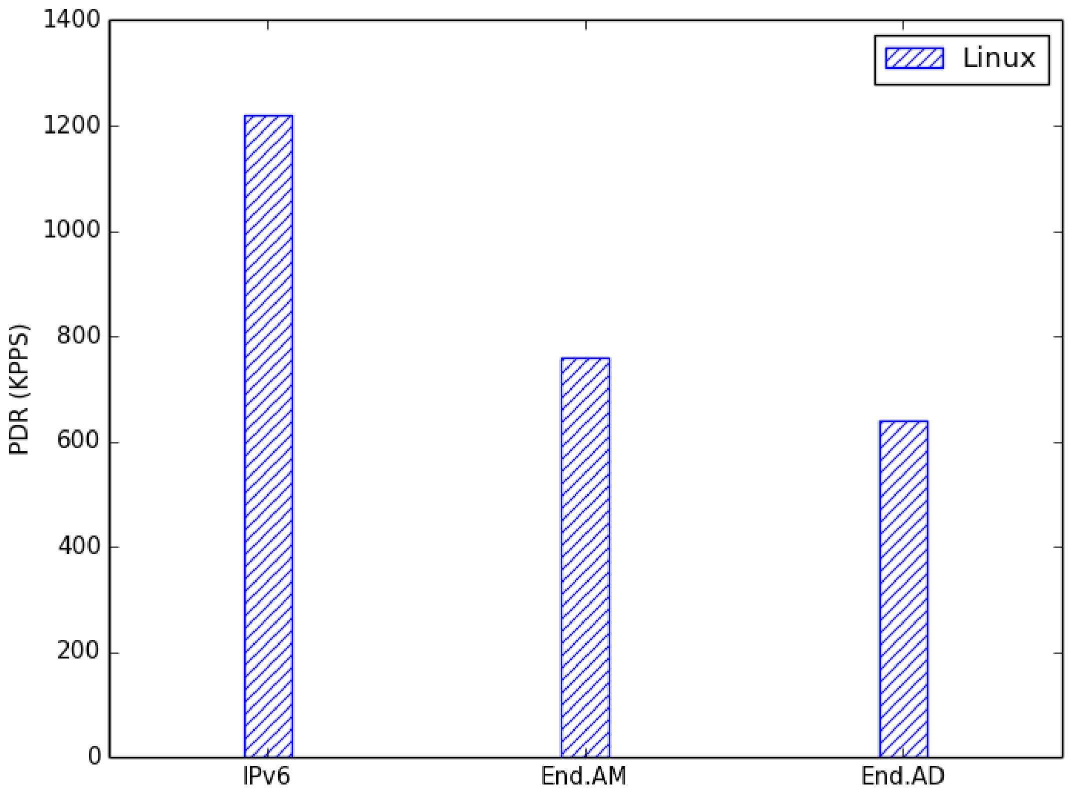


Figure 5.8: Performance measurements for Exp5.

5.5.1 Analysis of the performance issues of SRv6 cross-connect behaviors in Linux

The *End.X* and *End.DX6* behaviors exhibit poor performances compared to the other SRv6 *endpoint* behaviors as witnessed by the results reported in Table 5.1, and shown in Figures 5.5 and 5.7. The two behaviors share the same logic, they perform different variations of Layer 3 cross-connect to an adjacency set by the control plane. However, the current implementations of these two behaviors in Linux are not fully compliant with their specifications in the SRv6 network programming document [34]. The document specifies that SRv6 cross connect behaviors are used to cross connect packet to the next hop through a specific interface. However, the current implementations instead uses a layer3 next-hop, defined by the control plane, to forward the packet.

In the Linux kernel, when a packet has to be forwarded, the routing subsystem needs to find a route in the routing tables and returns a structure rt6_info as results of the lookup. When the next-hop of a given packet is already known, the lookup in the routing tables should be avoided. In facts, caches are widely used to avoid a lookup in the routing tables and further memory allocations for each packet if the next hop is known. However, in the implementation of *End.X* and *End.DX6* behaviors, the caches are not used and a structure rt6_info is allocated for each packet to emulate the lookup process. The memory allocation has a huge performance impact leading to such performance drop. To fix the poor performance of the cross-connect behaviors we extended their implementation in the Linux kernel to allow forwarding packets based on a outgoing interface instead of the next-hop. We implemented a new kernel function, named seg6_xcon6, which is called by the *End.X* and *End.DX6* to cross-connect the IPv6 packet to a given interface. We extended the same logic to the SRv6 *End.DX4* behavior by implementing another kernel function, named seg6_xcon4, which is called by the *End.DX4* to to cross-connect the IPv4 packet to a given interface. The fixes for the cross-connect behaviors are open source and available on GitHub [109].

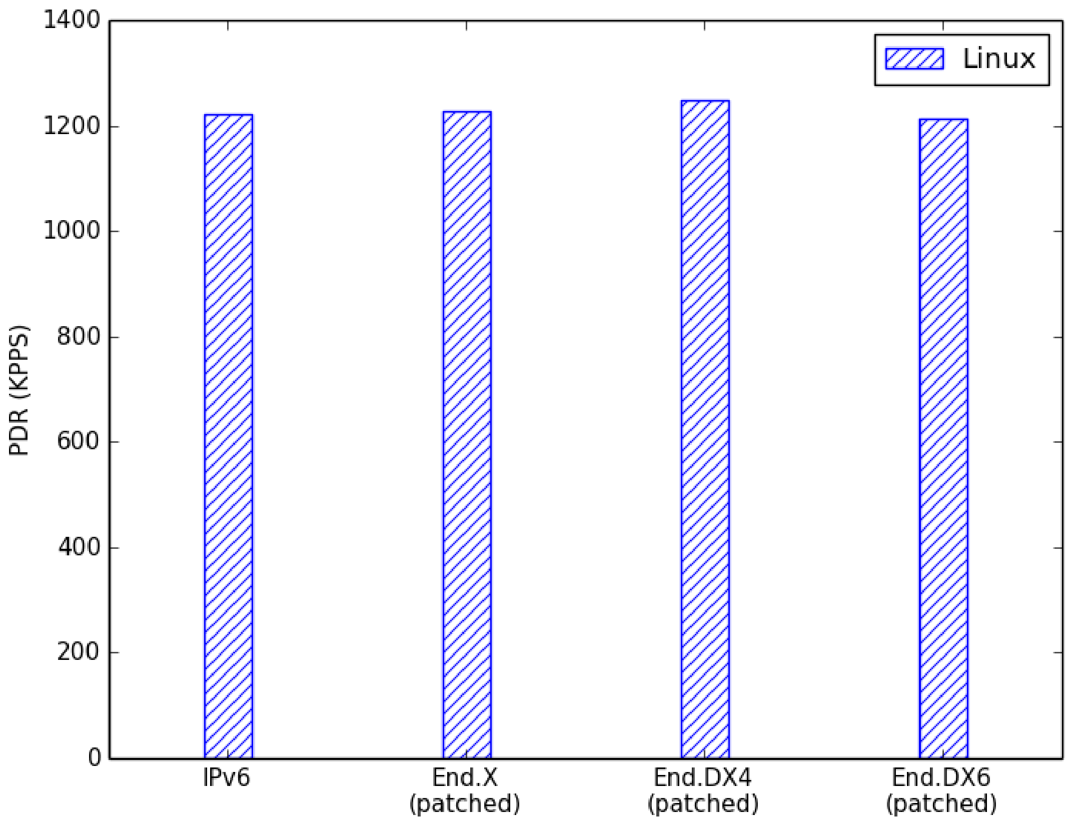


Figure 5.9: Performance measurements for Exp6.

In the Exp6 experiment category, we evaluated the performance of the patched Linux kernel for the *End.X*, *End.DX4* and *End.DX6*. The results of the experiment are reported in the Exp6 section of Table 5.1 and shown in Figure 5.9. The results show a noticeable improvement in the cross-connect behaviors. The *End.X* and *End.DX6* can forward $\approx$1233 and $\approx$1208 Kpps which is around 17 times faster with respect to the poor performance reported previously. The patched kernel has shown as well a performance improvement for the *End.DX4* behavior of around $\approx$300 Kpps.

5.6 Performance evaluation of SRv6 user-space packet forwarders.

The concept of implementing user-space software packets forwarders has gained a lot of attention from both industry and the research community. The idea is to map directly the network interface card (NIC) into user-space by bypassing the underlying kernel. These techniques relies on a sort of direct memory access (DMA) ring buffers [110] shared between the NIC driver and user-space, where the NIC driver writes directly to the ring buffer and the user-space packet forwarder reads directly from there. Netmap [111] and Data Plane Development Kit (DPDk) [47] are examples of such kernel by-passing mechanisms. DPDK is currently the most widely deployed kernel by-passing mechanism, supported in several NICs from various NIC vendors [112]. Several framework for fast packet I/O are built by leveraging DPDK such as Virtual Packet processor (VPP) [21] and Network Function Framework for Go (NFF-GO) [113].

In this section, we evaluate the forwarding performance of VPP. We focus on the SRv6 behaviors and compare their performance in VPP and Linux kernel. Firstly, we introduce the architecture of VPP in Section 5.6.1. Then, we carry out several experiments to evaluate the VPP forwarding performance (Section 5.6.2).

5.6.1 SRv6 support in VPP

VPP is a open source virtual switch and router that provides a high performance forwarder that can run on commodity CPUs. VPP is a very flexible and modular framework that allows the addition of new *plugins* without the need to change the core kernel code. The packet processing architecture of VPP consists of graph nodes that are composed together. Each graph node performs one function of the processing stack such as IPv6 packets input (*ip6-input*), or IPv6 FIB look-up (*ip6-lookup*). The composition of the several graph nodes of VPP are decided at runtime. Figure 5.10 shows an example of VPP packet processing graph running on top on DPDK. VPP also supports batch packet processing [114], a technique that allows the processing of a batch of packets by one VPP graph nodes before passing them to the next node. This technique improves the packets processing performance by leveraging the CPU instruction cache. The analysis of VPP performance is reported in [98] [114]. The main challenge to use VPP for high performance forwarding is that the server NIC must support DPDK. Also, the learning curve for using and writing code in VPP is quite steep which can be a challenge for some researchers.

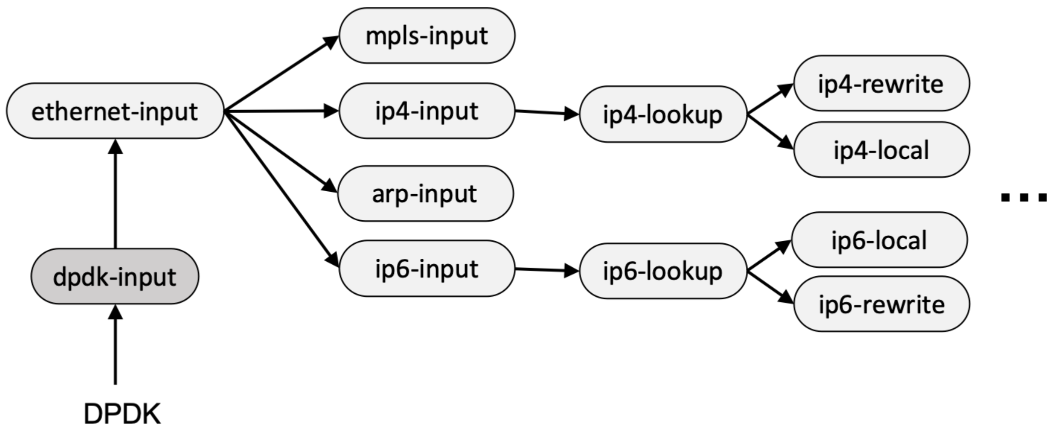


Figure 5.10: VPP packet processing architecture

SRv6 capabilities were introduced in the 17.04 release of VPP. Most of the SRv6 endpoint behaviors defined in [34] are supported (e.g. *End*, *End.X*, *End.DX2*, *End.DX4*, *End.DX6*, *End.DT4*, *End.DX6*). These behaviors are grouped by the endpoint function type and implemented in dedicated VPP graph nodes. For example, all the *encap* functions share one single graph node, while the *End* and *End.X* functions are implemented in another VPP graph node.

The SRv6 graph nodes perform the required SRv6 behaviors as well the IPv6 processing (e.g. decrement Hop Limit). When an SRv6 segment is instantiated, a new IPv6 FIB entry is created for the segment address that points to the corresponding VPP graph node. SRv6 segments are allocated off the global IPv6 address space. An API was added to allow developers create new SRv6 endpoint behaviors using the VPP plugin framework. In this way, a developer can focus on the actual behavior implementation while the segment instantiation, listing and removal are performed by the existing SRv6 code. Finally, the SR proxy behaviors were introduced as VPP plugins in release 18.04 [41].

5.6.2 SRv6 performance: VPP vs Linux kernel

In this section, we compare the forwarding performance of the Linux kernel and VPP. We used the same testbed described in Section 5.4. We carried out several experiments to compare the forwarding performance of VPP and the Linux kernel. The results of the experiments are available on GitHub [108] and reported in Table 5.2, where we show the PDR of each forwarding behavior (VPP and Linux), the line rate of each experiment, and the ratio versus the line rate of each forwarding behavior. For each experiment, we computed the line rate, which depends on the size of the packets as shown in Equation 5.1.

$$PacketLineRate = LineSpeed/[8 * (Size + Overhead)] \tag{5.1}$$

Where *LineSpeed* is the physical bit rate (e.g. $10 * 10^9$ for 10GbE, *Size* is the frame size at Ethernet level, the *Overhead* for the Ethernet frames is 24 bytes (4 for CRC, 8 for preamble/SFD and 12 for the inter frame gap). For a 10GbE interface and an IP packet of 64 bytes, the *Size* is 78 bytes and the packet line rate is $\approx$12255 kpps.

We classified the experiments in different categories as follows:

- Exp7: Plain IP forwarding.
- Exp8: SRv6 transit behaviors.
- Exp9: SRv6 endpoint behaviors with no decapsulation.
- Exp10: SRv6 endpoint behaviors with decapsulation and lookup in a specific routing table.
- Exp11: SRv6 endpoint behaviors with decapsulation and cross-connect through an outgoing interface.

Experiment	Behavior	VPP	Linux	Line rate
Exp7	IPv4	12252.63	1430.38	12253
		99.99%	11.67%	
	IPv6	11327.51	1221.05	
		92.44%	9.97%	
Exp8	T.Insert	7387.16	1039.29	8802
		83.92%	11.81%	
	T.Encaps	7709.82	978.13	
		87.58%	11.11%	
	T.Encaps.L2	8052.85	828.89	
		91.17%	9.41%	
Exp9	End	6867.59	900.52	6868
		99.99%	13.11%	
	End.T	6867.59	979.25	
		99.99%	14.25%	
	End.X	6867.59	1228.71	
		99.99%	17.89%	
Exp10	End.DT4	6867.58	1027.00	6868
		99.99%	14.95%	
	End.DT6	6867.59	960.06	
		99.99%	13.97%	
Exp11	End.DX2	6867.58	1299.15	6868
		99.99%	18.92%	
	End.DX4	6867.59	1248.29	
		99.99%	18.17%	
	End.DX6	6867.59	1213.06	
		99.99%	17.66%	

Table 5.2: Performance measurements for VPP vs Linux (Kpps).

Exp7: Plain IP forwarding

In the Exp7 experiment category, we measure the performance of plain IP forwarding for both the Linux kernel and VPP, which allows to understand the base forwarding performance of the two forwarding engines prior to measuring the SRv6 performance. We use a test packet of size 64 bytes. In case of IPv4, we used an IPv4 packet composed of IPv4 header (20 bytes), UDP header (8 bytes) and payload (36 bytes). In case of IPv6, we used an IPv6 packet composed of IPv6 header (40 bytes), UDP header (8 bytes), and payload (16 bytes). The computed line rate for Exp7 is $\approx$12253 kpps. The results are reported in the Exp7 section of Table 5.2, and shown in Figure 5.11.

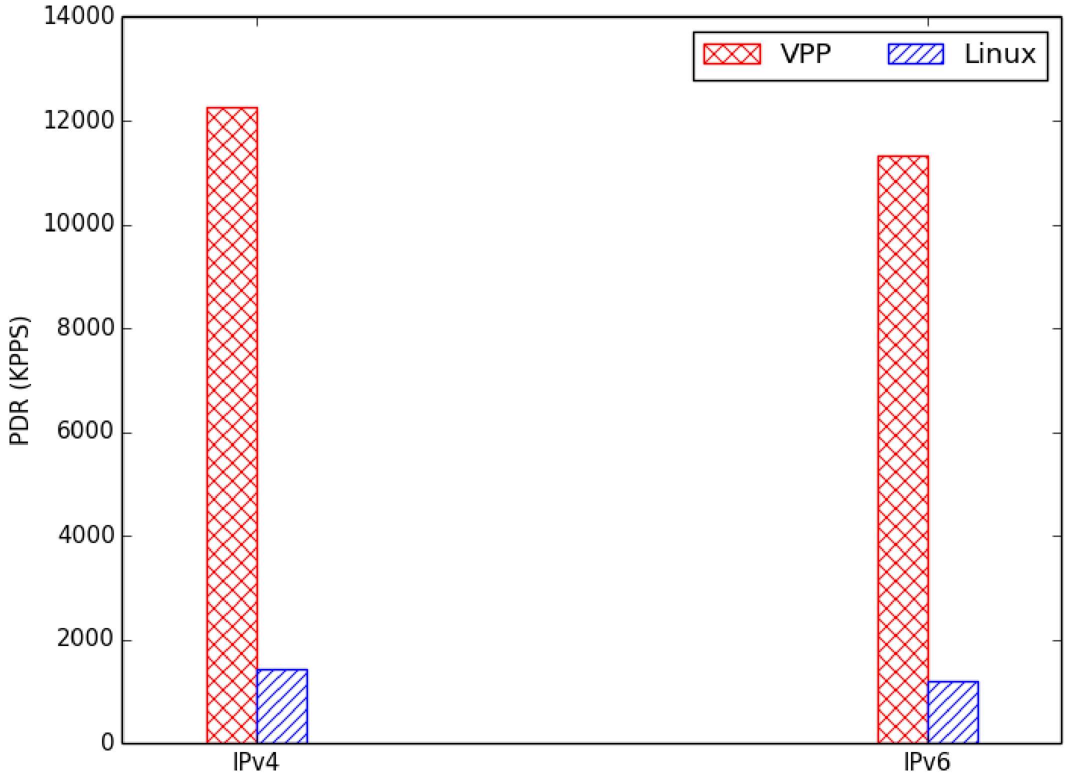


Figure 5.11: Performance measurements for Exp7.

The experiment has shown the capabilities of VPP to forward packets at much higher rate compared to the Linux kernel. The reason is that VPP is a specialized software for packet forwarding while the Linux kernel is a general purpose kernel that can be used for packets forwarding. In addition, VPP utilizes several forwarding optimization techniques such as batch packet processing that allows to process a vector of packets at once unlike the Linux kernel which processes packets in one-by-one manner. In addition, VPP uses DPDK [47] to acquire packets from the NIC, which allows to bypass all the kernel processing and focus just on packets processing functions. The results show that the IPv4 stack performs better compared to the IPv6 stack, in both VPP and Linux. In VPP, the IPv6 stack can forward ≈11327 Kpps which is 7.5% less than the ≈12252 Kpps that the IPv4 stack can forward. In Linux, the IPv4 stack can forward ≈1430 Kpps, while the IPv6 stack can only forward ≈1221 Kpps. The IPv4 stack has better performance as it has been used for production long time before IPv6, so it gets updated and optimized more times with respect to the IPv6 stack.

Exp8: SRv6 transit behaviors

In the Exp8 experiment category, we measure the performance of the SRv6 *T.Insert*, *T.Encaps* and *T.Encaps.L2* transit behaviors. We use the same IPv6 packet described in Exp7. The line rate for this experiment is ≈8802 Kpps. The results are reported in the Exp8 section of Table 5.2, and shown in Figure 5.12.

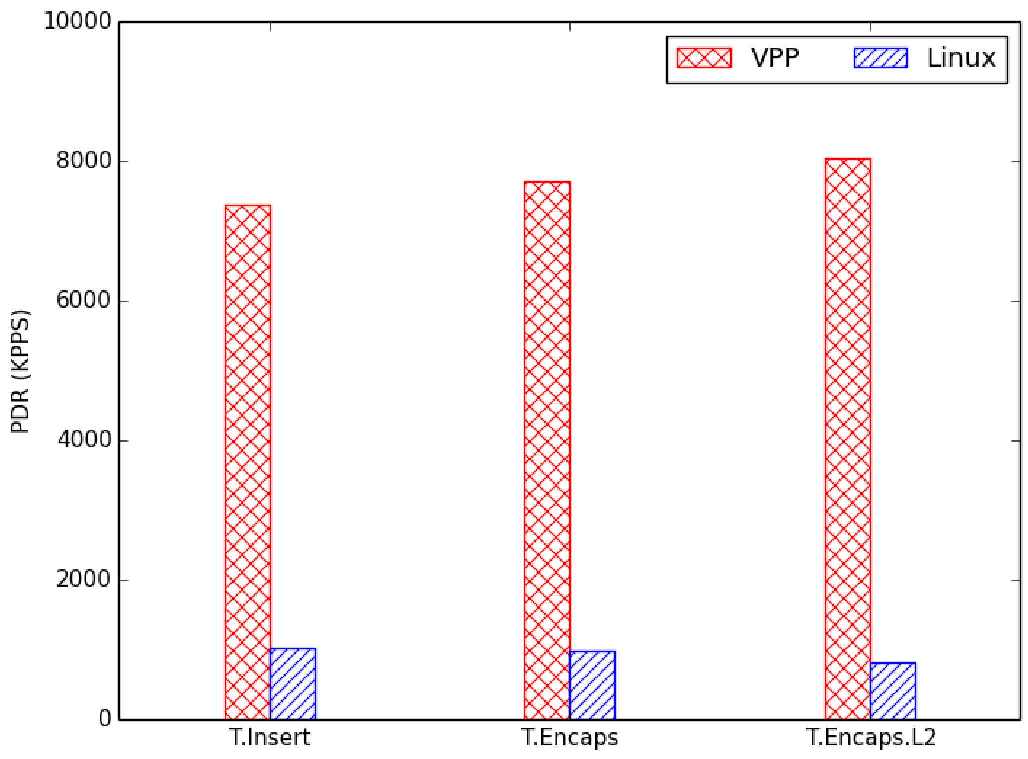


Figure 5.12: Performance measurements for Exp8.

The results show that the SRv6 transit behaviors introduce an overhead in the packets forwarding path, reducing the forwarding capability. This is the result of several operations that the SRv6 transit behaviors have to perform on each received packet as described for the experiment category Exp1. In VPP, the SUT machine is able to forward ≈7378, ≈7709, and ≈8052 Kpps respectively for the *T.Insert*, *T.Encaps*, and *T.Encaps.L2* behaviors. The *T.Insert* behavior has the lowest performance among the SRv6 transit behaviors because it implies inserting an SRH between the IPv6 header and transport layer header. This requires two memory copy operations: the first to move the IPv6 header to create the space required for the SRH insertion, and the second is to copy the actual SRH into the created space. The other SRv6 transit behaviors does not require the first memory copy operation as the SRv6 encapsulation is copied directly in the memory preceding the packet. In case of Linux, the SUT machine is able to forward ≈1039, ≈978, and ≈828 Kpps respectively for the *T.Insert*, *T.Encaps*, and *T.Encaps.L2* behaviors.

Exp9: SRv6 endpoint behaviors without decapsulation

In the Exp9 experiment category, we evaluate the performance of three SRv6 endpoints behaviors: *End*, *End.T*, and *End.X*. These behaviors do not require removing the SRv6 encapsulation from packets. We use an SRv6 packet comprised of an outer IPv6 header (40 bytes), an SRH containing two SRv6 SIDs (40 bytes) and inner packet (64 bytes). The SRv6 packet has a total length of 144 bytes which allows to send packets at line rate of ≈6868 Kpps. The results are reported in the Exp9 section of Table 5.2, and shown in Figure 5.13.

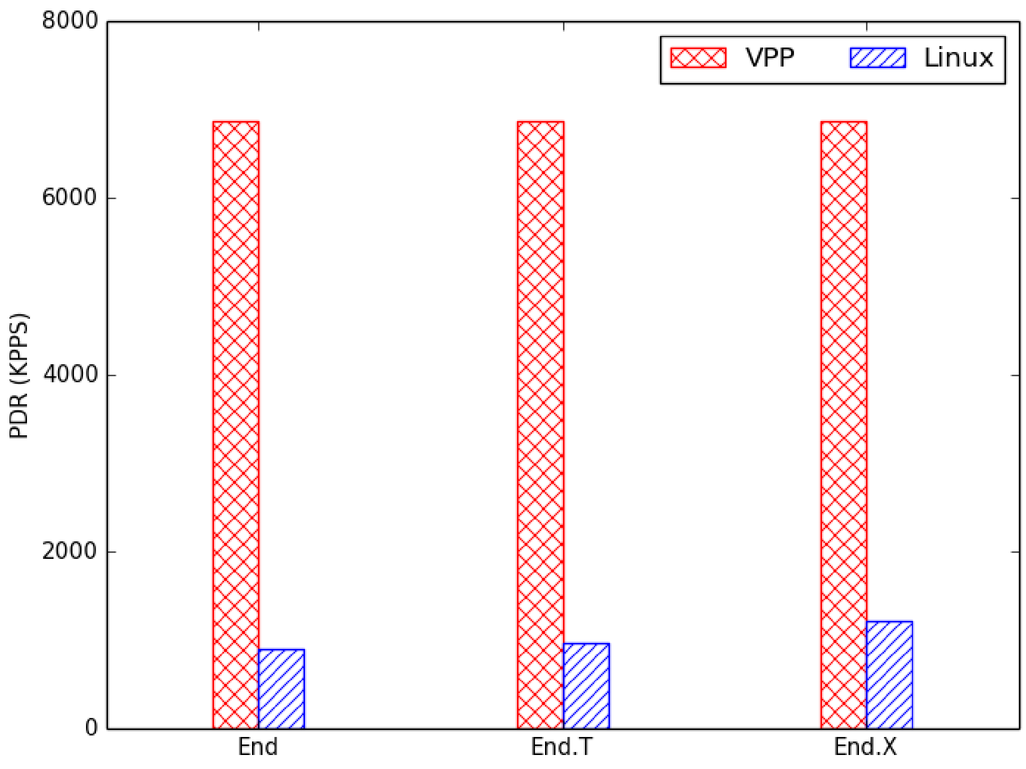


Figure 5.13: Performance measurements for Exp9.

In VPP, the *End*, *End.T* and *End.X* behaviors are performing at Line rate, which means that the SUT machine is able to forward all the received packets (≈6868 Kpps). In Linux, the *SUT* machine is able to forward ≈900, ≈979 and ≈1228 Kpps respectively for the *End*, *End.T* and *End.X* behaviors.

Exp10: SRv6 endpoint behaviors with decapsulation and lookup in a specific routing table

In the Exp10 experiment category, we evaluate the performance of two SRv6 endpoints behaviors: *End.DT4* and *End.DT6*. These behaviors decapsulate the SRv6 information and perform routing lookup in a specific routing table associated with the SRv6 SID. We use the same SRv6 packet format used for Exp9. The line rate for this experiment is the same as in Exp9 (≈6868 Kpps) as we use the same packet format. The results are reported in the Exp10 section of Table 5.2, and shown in Figure 5.14.

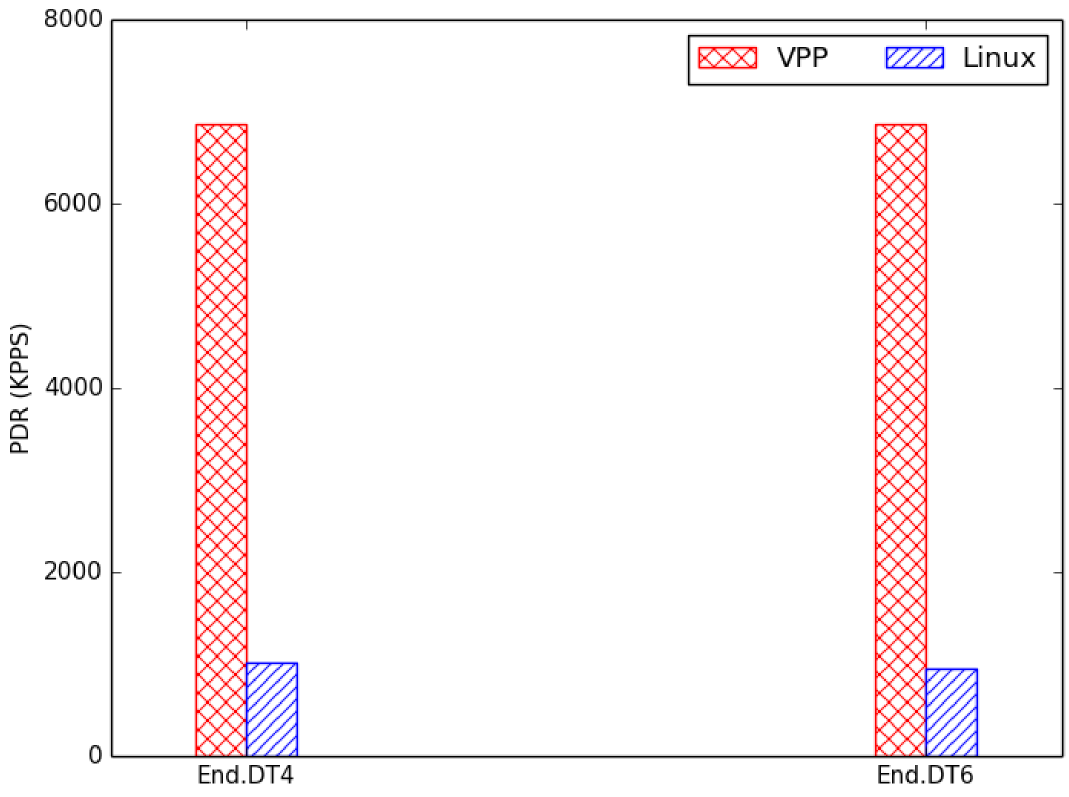


Figure 5.14: Performance measurements for Exp10.

In VPP, the *End.DT4* and *End.DT6* behaviors are performing at line rate. In Linux, the *SUT* machine is able to forward ≈1027 and ≈960 Kpps respectively for the *End.DT4* and *End.DT6* behaviors.

Exp11: SRv6 endpoint behaviors with decapsulation and cross-connect through an outgoing interface

In the Exp11 experiment category, we evaluate the performance of three SRv6 endpoints behaviors: *End.DX2*, *End.DX4*, and *End.DX6*. These behaviors remove the SRv6 information and send the inner packet towards the final destination by doing cross-connect to an outgoing interface associated with the SRv6 SID. We use the same SRv6 packet format of Exp9. The line rate for this experiment is the same as in Exp9 (≈6868 Kpps). The results are reported in the Exp11 section of Table 5.2, and shown in Figure 5.15.

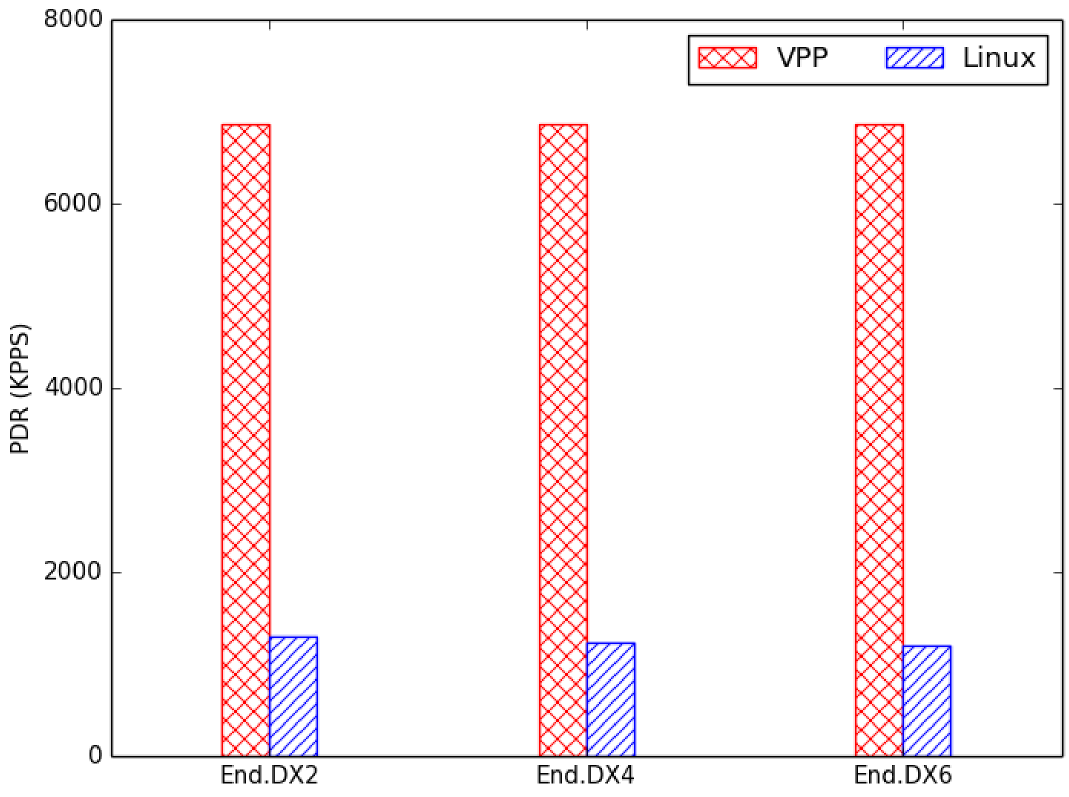


Figure 5.15: Performance measurements for Exp11.

In VPP, the *End.DX2*, *End.DX4* and *End.DX6* behaviors are performing at Line rate (≈6868 Kpps). In Linux, the *SUT* machine is able to forward ≈1299, ≈1248 and ≈1213 Kpps respectively for the *End.DX2*, *End.DX4*, and *End.DX6* behaviors.

5.7 Related work

The era of SDN and NFV results in revolution in the development of software forwarding elements (e.g., virtual routers and switches). These forwarding elements gives a great flexibility to add new data plane features, at very fast pace, compared to previous hardware based forwarders. Software forwarders utilizes commodity hardware's CPU for forwarding operations. However, the software forwarder performance on commodity CPUs require careful measurement and analysis as such CPUs were not designed specifically for traffic forwarding. In order to address these needs, several frameworks have been developed. However, none of the works found in literature have fully addressed the performance of SRv6 data-plane implementation either in Linux kernel and other software router implementations (e.g., VPP).

DPDK [47] is the state of the art technology for accelerating the virtual forwarding elements. It bypasses the kernel processing and balances the incoming flow of packets over all the CPU cores and processes packets by batches to make a better use of the CPU cache. In [115], the authors presented an analytical queuing model to evaluate the performance of a DPDK-based vSwitch. The authors studied several characteristics of DPDK such as average queue size, average sojourn time and loss rate under different arrival loads measured in packet per second (pps).

In [116], the performance of several virtual switch implementations including Open vSwitch (OVS) [57], SR-IOV and VPP are investigated. The work focuses on the NFV use-cases where multiple VNFs run in x86 servers. The work shows the system throughput in a multi-VNF environment. However, this work considers only IPv4 type of traffic and does not address SRv6 related performance. In [117], this work has been extended by replacing OVS with OVS-DPDK [118], which promises to significantly increase the I/O performance for virtualized network functions. They use DPDK-enabled VNFs and show how OVS-DPDK compares from a throughput perspective to SR-IOV and VPP as the number of VNFs is increased under multiple feature configurations. However, the work still considers only plain IPv4 forwarding.

In [114], the authors explain the main architecture principles and components of VPP including: vector processing, kernel bypass, packets batch processing, multi-loop, branch-prediction, function flattening and direct cache access. To validate the high speed forwarding capabilities of VPP, the authors have reported some performance measurements such as packet forwarding rate for different vector sizes (i.e, number of packets processed as a single batch), the impact of multi-loop programming practice on the per-packet processing cost as well the variation of the packet processing rate as a function of the input workload process. However, this work does not analyse VPP performance for different types of traffic such as IPv6 or SRv6.

Open Platform for NFV Project (OPNFV) [119] is a Linux foundation project which aims to provide carrier-grade, integrated platform to introduce quickly in the industry new products and services. The NFVbench [120] toolkit, developed under the OPNFV umbrella, allows developers, system integrators, testers and customers to measure and assess the L2/L3 forwarding performance of an NFV-infrastructure solution stack using a black-box approach. The toolkit is agnostic of the installer, hardware, controller or the network stack used. VSPERF [121] is another project within the OPNFV specialized for benchmarking virtual switch performance. VSPERF reported results for both VPP and OVS which are based on daily executed series of test-cases [122].

The FD.io project has released a technical paper [123] for analysing the performance of several data plane implementations such as DPDK, VPP, OVS-DPDK. The work reports a comparison between DPDK L2 forwarding, OVS-DPDK L2 Cross-Connect, VPP L2 Cross-Connect and VPP IPv4 forwarding in terms of throughput measured in pps. The FD.io Continuous System Integration and Testing (CSIT) project released a report characterizing VPP performance [98]. The report describes a methodology to test VPP forwarding performance for several test cases including: L2 forwarding, L3 IPv4 forwarding, L3 IPv6 forwarding as well as some SRv6 behaviors. Regarding the latter, the report shows the performance of SRv6 *T.Encaps*, *End.AD*, *End.AM* and *End.As* behaviors in VPP. However, the report does not cover the performance of the rest of SRv6 *transit* and *endpoint* behaviors in VPP. Moreover, the framework is designed only for VPP and can not be used to measure the performance of other forwarder (e.g., Linux Kernel).

The performance of some SRv6 behaviors is reported in [35], [36]. The work has mainly focused on the SRv6 *transit* behaviors, namely *T.Insert* and *T.Encaps*. The reported results show the overhead introduced by applying the SRv6 encapsulation to IPv6 traffic. However, the performance reported in this work can be considered out-dated as it considered the SRv6 implementations in Linux kernel 4.12 release. Moreover, the work does not report the performance of any SRv6 *endpoint* behavior as they were not supported in the kernel by that time.

Chapter 6

Conclusions

Telecommunication networks are continuously evolving to satisfy the bandwidth and quality-of-service demands of the growing number of connected users and devices. Technologies such as Software Defined Networking (SDN) and Network Function Virtualization (NFV) play a major role in such evolution. They allow networks to offer advanced services beyond mere packet forwarding. Yet, service function chaining (SFC) is one of the most challenging use-cases in telecommunication networks. In this thesis, we provided a scalable solution for the SFC problem, which leverages the IPv6 segment routing (SRv6) architecture. Being SRv6-based, our approach does not need to maintain per-chain state information in the network fabric. Our solution is thus more scalable than existing traditional alternatives.

We addressed an important problem faced by researchers when they need to implement a new networking feature in the Linux kernel. As the available documentation of the Linux kernel is either outdated or incomplete, we provided a tutorial on Linux kernel networking and on the SRv6 implementation. The tutorial (Appendix A) is based on tracing and deep walk through the Linux kernel source code and all available documentation. In this tutorial, we explained the main data-structures of the Linux kernel IPv6 network stack such IPv6 packet and IPv6 FIB (Forwarding Information Base) entry. In addition, we discussed the different sub-systems of the IPv6 protocol stack in the Linux kernel. Also, we presented the Netfilter framework architecture and how it can be used extend the Linux network stack by attaching kernel modules. Finally, we provided a detailed discussion of the SRv6 support across the various releases of the Linux kernel.

In Chapter 3, we designed a scalable SRv6-based SFC architecture. We defined both the data and control planes aspects of our SFC architecture. In this architecture, we differentiated between two types of VNFs: SRv6-aware and SRv6-unaware VNFs. The SRv6-aware VNFs can be integrated directly in our SFC architecture. For the SRv6-unaware VNFs, we defined several SRv6-proxy behaviors as a solution to include these VNFs into our architecture. The proxy behaviors include static, dynamic, shared-memory and masquerading proxies. We provided an open source implementation for these proxy behaviors. The implementation is in the form of Linux kernel module. We also contributed to the work of integrating the proxy implementation in the mainline of the Linux kernel.

In Chapter 4, we took a forward-looking approach by designing and implementing several SRv6-aware VNFs. These VNFs can be integrated directly into our SRv6-based SFC architecture without the need of the SRv6 proxy. They can also perform advanced SRv6 actions such as skipping the next SID in the segment list, so that it is possible to operate a “branching” instead of the usual linear exploration of the VNF chain, when some conditions on the packet are met. We implemented several SRv6-aware VNFs, including SERA, SR-Snort, SR-nftables and SR-tcpdump. SERA is an SRv6-aware firewall capable of applying normal firewall behaviors to packets with SRv6 information. It can also perform stateless SRv6-specific actions on packets. We provided an open source implementation of SERA which extends the commonly used iptables firewall. In SR-Snort, we extended the widely deployed open source IDS/IPS (Snort) to be SRv6-aware. SR-nftables, is another SRv6-aware network function which extends the next generation Linux firewall (nftables). SR-tcpdump is a tool that allows analyzing and debugging SRv6 traffic. It extends the tcpdump tool with a dissector for SRv6. The different SRv6-aware network functions developed for this thesis allow to build a fully automated SRv6-based SFC architecture. We contributed our implementations of SRv6-aware network functions to several open-source projects to be used by network operators as well as other researchers. Several parts of these implementations are currently integrated into the mainline of the Linux kernel, like the iptables and nftables components and the tcpdump tool.

In Chapter 5, we designed a performance evaluation framework for SRv6 (SRPerf). The main motivation for designing such framework is that SRv6 has been deployed in several service providers networks and data-centers. Hence, it is critical to assess non-functional properties of SRv6 such as scalability and fault tolerance. SRPerf complies with the network benchmarking guidelines defined in RFC2544. SRPerf supports SRv6 and plain IP forwarding. It can use Linux networking stack or VPP (Vector Packet Processing) as forwarding engine but new forwarding engines can be easily added. It reports different throughput measures such as NDR

(No Drop Rate), PDR (Partial Drop Rate). We have used SRPerf to evaluate the performance of the SRv6 implementation in the Linux kernel and VPP. The framework allowed us to identify some performance issues of the SRv6 implementation which we have fixed in new revisions.

Appendix A

Linux kernel networking

The Linux kernel is the largest open source software project on the planet [124]. It was originally developed by Linux Torvalds in 1991 [125]. It has been very successful as an operating system and a software project [126] [127]. Release 4.13 of the Linux kernel has over 24 million lines of code, and has been contributed by around 1681 developers [128]. Linux is the operating system for over 95% of the top one-million domains and 75% of cloud-enabled enterprises [129]. The Linux kernel community provides different learning resources for developers and users including LWN.net [130], documentation [131], man pages [132], and mailing list archives [133]. However, the learning curve is very steep due to out-dated and very generic, or sometimes fragmented information between different learning resources.

In this appendix, we discuss the networking subsystem of the Linux kernel, as it is the platform used for all the implementations throughout the thesis work. The content is based on tracing and deep walk through the Linux kernel source code and all available documentation. This appendix is a guide for understanding the SRv6 implementations in Linux kernel. In Section A.1, we explain the main building blocks of the the networking subsystem of the Linux kernel focusing on the IPv6 protocol stack. In Section A.2, we introduce the netfilter framework. Finally, Section A.3 provides the details of SRv6 support in Linux.

A.1 Linux networking subsystem

In this section we explain in detail the various components of the IPv6 protocol stack of the Linux kernel shown in Figure A.1.

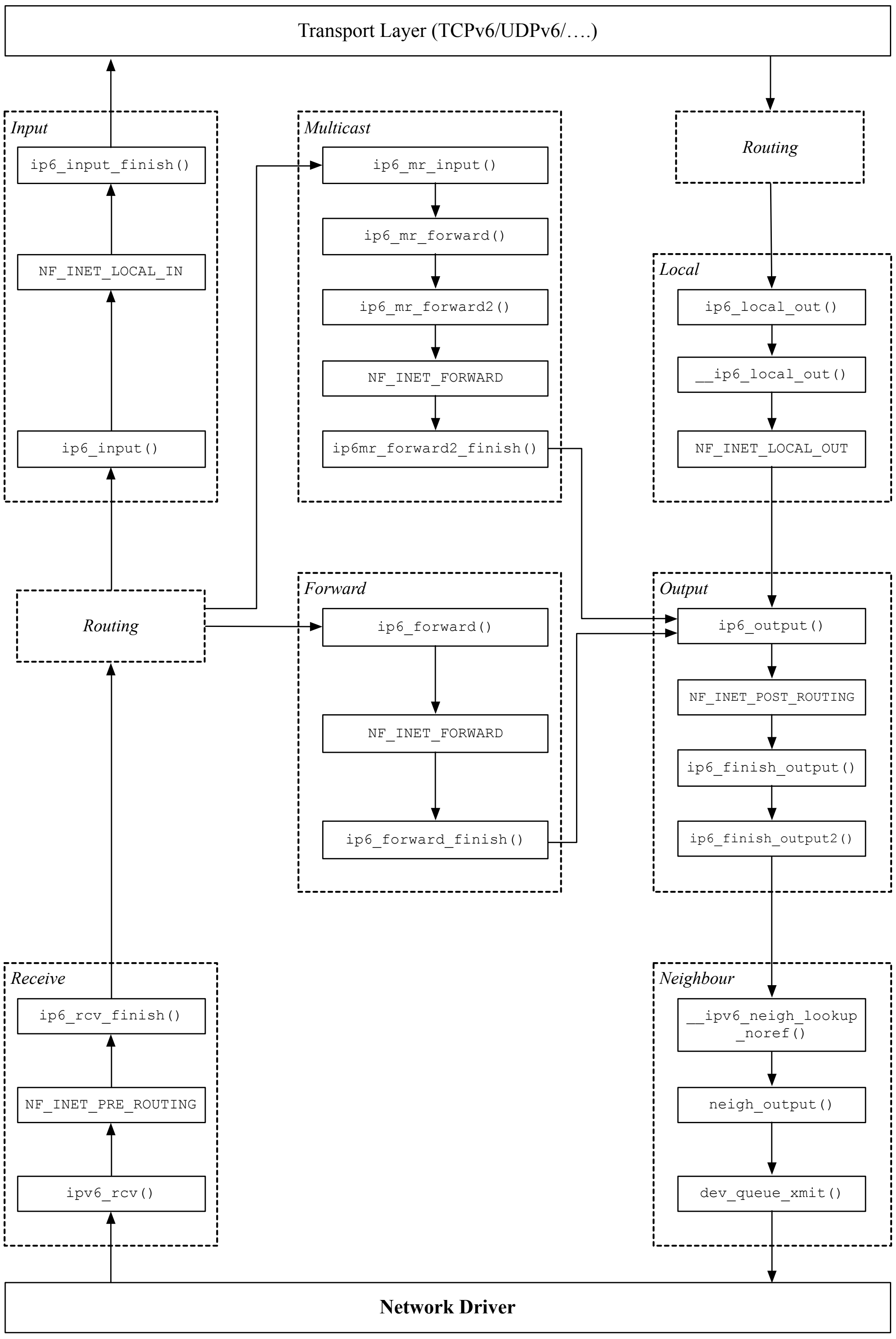


Figure A.1: IPv6 protocol stack in the Linux kernel

A.1.1 IPv6 packet

In the Linux kernel, IPv6 packets just as any other type of packets are represented by struct sk_buff, which is often referred to as skb [134]. Listing A.1 shows few of the skb structure members. These fields are used as follows:

Listing A.1: Socket buffer structure

```
struct sk_buff {
    .....
    struct sock         *sk;
    struct net_device   *dev;
    unsigned int        len;
    unsigned long       _skb_refdst;
    __u8                cloned:1
    __u8                encapsulation:1;
    __be16              inner_protocol;
    __u16               inner_transport_header;
    __u16               inner_network_header;
    __u16               inner_mac_header;
    __be16              protocol;
    __u16               transport_header;
    __u16               network_header;
    __u16               mac_header;
    sk_buff_data_t      tail;
    sk_buff_data_t      end;
    unsigned char       *head;
    unsigned char       *data;
    .....
};
```

- sk: The socket that owns the skb.
- dev: The network interface associated with the skb. It can be the network interface on which the packet arrives, or the network interface on which the packet is going to be sent.
- len: The total length of the IPv6 packet in bytes. It covers the packet headers and payload.

- _skb_refdst: A pointer to struct dst_entry that has the route decision for the skb. The dst_entry has two callback functions (input and output) to handle the skb in both receive (Rx) and transmit (Tx) path. When the routing subsystem finds a match between an skb and a routing entry, it sets the _skb_refdst to point to the dst_entry of the matched routing entry.
- cloned: Buffer might have been cloned.
- encapsulation: A single-bit field indicating whether the values of fields inner_protocol, inner_mac_header, inner_network_header, and inner_transport_header are valid or not.
- inner_protocol: Internet protocol of the inner packet.
- inner_transport_header: The offset of outer transport header in the skb's data.
- inner_network_header: The offset of inner network header in the skb's data.
- inner_mac_header: The offset of the inner mac header in the skb's data.
- protocol: The internet protocol of outer packet.
- network_header: The offset of the outer network header in the skb's data.
- mac_header: The offset of the outer mac header in the skb's data.
- tail: A pointer to the tail room of skb.
- end: A pointer to the end of skb.
- head: A pointer to the head room of skb.
- data: A pointer to the data of skb.

A.1.2 IPv6 FIB entry

The forwarding information base (FIB) is composed of a set of entries, knows as FIB entries, that contain the necessary information to make a forwarding decision on a particular packet [135]. In the Linux kernel, IPv6 FIB entries are represented by struct rt6_info [136], as shown in Listing A.2.

Listing A.2: IPv6 routing entry structure

```
struct rt6_info {
    struct dst_entry    dst;
    struct fib6_table   *rt6i_table;
    struct in6_addr     rt6i_gateway;
    ...
    struct rt6key       rt6i_dst;
    u32                 rt6i_flags;
    struct rt6key       rt6i_src;
    ...
    struct inet6_dev    *rt6i_idev;
    ...
    u32                 rt6i_metric;
    u32                 rt6i_pmtu;
    ...
    u8                  rt6i_protocol;
};
```

- dst: A struct dst_entry that holds the route information for those IPv6 packets that match with the FIB entry.
- rt6i_table: The routing table that contains the FIB entry. When inserting a new route entry, this member instructs the kernel on which table to insert a new entry.
- rt6i_gateway: The Next hop associated with a FIB entry.
- rt6i_dst: The IPv6 destination prefix of a FIB entry.
- rt6i_flags: The flags of the FIB entry.
- rt6i_src: Source prefix of the FIB entry. Used for source address based routing.
- rt6i_idev: The configuration of the interface associated with the FIB entry.
- rt6i_metric: The metric of the FIB entry.
- rt6i_pmtu: The path MTU associated with the FIB entry.
- rt6i_protocol: The source of the routing update that triggers the addition of the route entry. It can be an administrator (using iproute2), routing daemon (OSPF, BGP), or it can the kernel itself. A complete list of possible sources is defined in [137].

The struct dst_entry [138] is shown in Listing A.3. The input and output callbacks are used to handle IPv6 packets in the receive (*Rx*) and transmit (*Tx*) paths.

Listing A.3: Destination entry structure

```
struct dst_entry {
     ...
     int (*input)    (struct sk_buff*);
     int (*output)   (struct sk_buff*);
     ...
};
```

A.1.3 Manipulating IPv6 FIB

In Linux kernel, multiple events can trigger the creation, deletion, or update of an IPv6 FIB entry. Such events can be classified as are either *kernel-space* or *user-space*. Kernel-space events could be (re)configuring an interface, receiving neighbor discovery messages, receiving ICMP redirect messages. User-space events are usually generated by *iproute2* [75] (iproute2 is a set of user-space programs for interacting with the Linux networking) or a routing daemon such as OSPF [2] or BGP [1]. Considering the addition of a FIB entry, user-space added entries are passed to the kernel in the form of netlink [139] new route (RTM_NEWROUTE) message over a netlink socket. New route messages are handled in the kernel by rtnetlink_rcv() as shown in Figure A.2. Eventually the inet6_rtm_newroute() is invoked to add a new IPv6 FIB entry. The FIB configuration is extracted from the netlink message using rtm_to_fib6_config().

The process is performed in ip6_route_add() by calling ip6_route_info_create() to create the actual FIB entry. Finally the FIB entry is inserted into the FIB table using __ip6_ins_rt() function. The struct rt6_info is created by ip6_route_info_create(). In the ip6_route_info_create() the callback functions are assigned. The output callback for all IPv6 FIB entries is ip6_output(). The input callback is assigned as follows:

- ip6_input(): FIB entries with local address.
- ip6_mc_input(): FIB entries with multicast address.
- ip6_forward(): other FIB entries.

The routing subsystem is invoked per packet to match the skb against the FIB entries. Once an skb matches a FIB entry, the _skb_refdst is set to point to the struct dst_entry of the FIB entry. The *Rx* and *Tx* processing is done by invoking the input and output of the assigned dst_entry.

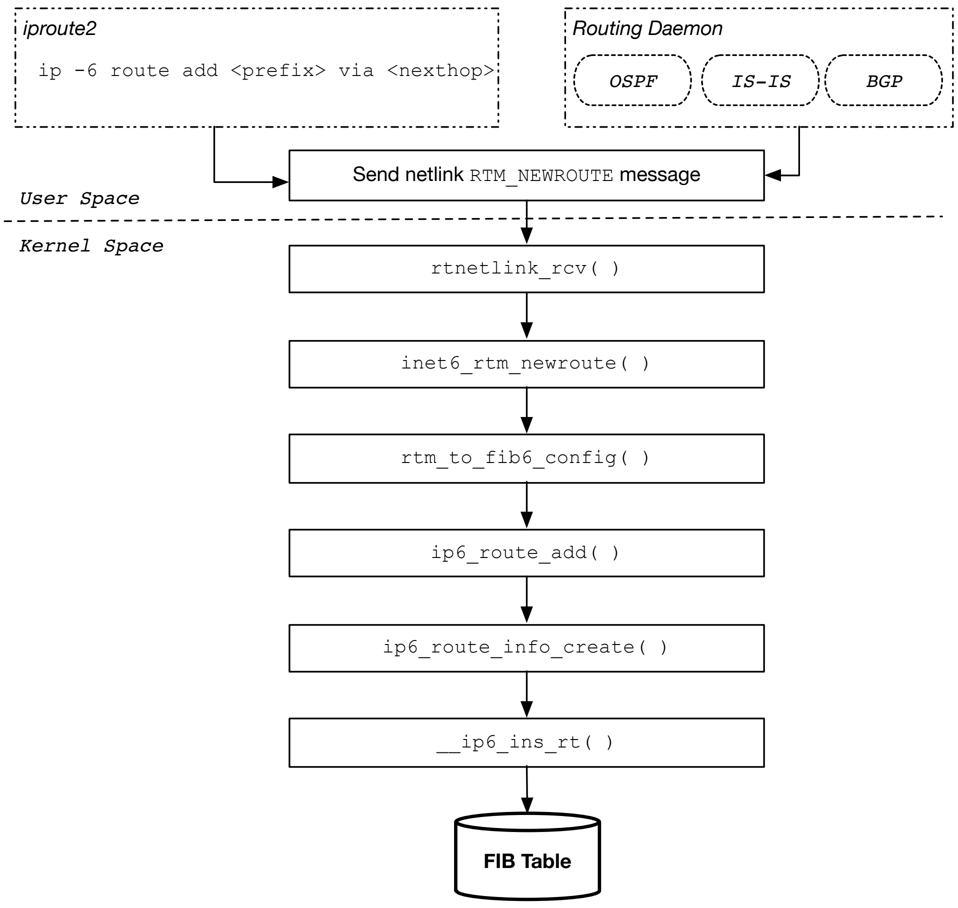


Figure A.2: Adding IPv6 FIB entries from user space

A.1.4 Linux Lightweight tunnels

The infrastructure to support lightweight tunnels (lwtunnel) was introduced in Linux kernel 4.3 [140]. Tunnels allow for scalable flow-based encapsulation (e.g., MPLS). The infrastructure allows to parse, dump, or store encap information for those lightweight tunnels. The encap information for such tunnels is associated with FIB entries. Two new callbacks are defined: lwtunnel_input(), lwtunnel_output(), which respectively override the input and output callbacks of a FIB entry. These callbacks can apply custom processing to IPv6 packets.

IP6-in-IP6 tunnels, SRv6 transit behaviors, SRv6 end behaviors are example implementations that use such callbacks. While lightweight tunnels were implemented with encapsulation in mind, the overriding input and output callbacks have no obligation to encapsulate the processed packet at all (e.g., the SRv6 end behaviors that do not encapsulate processed packets).

A.1.5 IPv6 protocol stack in Linux

The IPv6 protocol stack of the Linux kernel can be divided into eight main subsystems; *Receive*, *Routing*, *Input*, *Forward*, *Multicast*, *Local*, *Output* and *Neighbor*. As shown in Figure A.1, received IPv6 packets are passed by the network driver to the *Receive* subsystem. Then the *Routing* subsystem, based on the destination address or other attributes (e.g., source address), decides the next subsystem to process the packet. It may hand over the packet to the *Input*, *Forward*, or *Multicast* subsystem. The *Input* subsystem handles packets destined to a local unicast IPv6 address and hence delivers the packets to upper-layer protocols. The *Forward* subsystem handles packets to non-local addresses that should be forwarded (i.e., there is a route to the destination). The *Multicast* subsystem processes packets with an IPv6 multicast destination address. Packets processed by *Multicast*, *Forward*, and *Local* (handles locally generated IPv6 packets) subsystems are passed to the *Output* subsystem, which performs the final processing of the packet before handing it over to the *Neighbor* subsystem. For example, it checks whether a packet needs to be fragmented. Finally, the *Neighbor* subsystem pushes packets to the send queue of the network driver.

Receive subsystem

In the Linux kernel, the default handler for received IPv6 traffic (including multicast) is the ipv6_rcv() kernel function. It handles received ethernet frames that have an *ethertype* of ETH_P_IPV6 (0x86DD). Some sanity checks are performed on the received packet, such as verifying that the IPv6 header of a packet is not malformed, or that the source address of a received packet is not a multicast one [141]. In case a Hop-by-Hop options header is present, it must be the first one and will be parsed by ipv6_parse_hopopts(). After the packet passes all the sanity checks, the netfilter pre-routing (NF_INET_PRE_ROUTING) registered callbacks are invoked, if any (Section A.2). If the packet is not dropped in the pre-routing hook, then the ip6_rcv_finish() is invoked which performs lookup in the routing subsystem by calling ip6_route_input().

Routing subsystem

The *Routing* subsystem, shown in Figure A.3, is invoked by calling ip6_route_input(). It extracts the flow information (struct flowi6) from skb. Then it drops the dst reference count, if a reference was taken, by calling skb_dst_drop(). The IPv6 FIB lookup process is invoked by calling ip6_route_input_lookup(), which calls fib6_rule_lookup(). There are two different implementations of fib6_rule_lookup(): one for policy-based routing (PBR) enabled kernels [142], and the other for non-PBR enabled kernels [143].

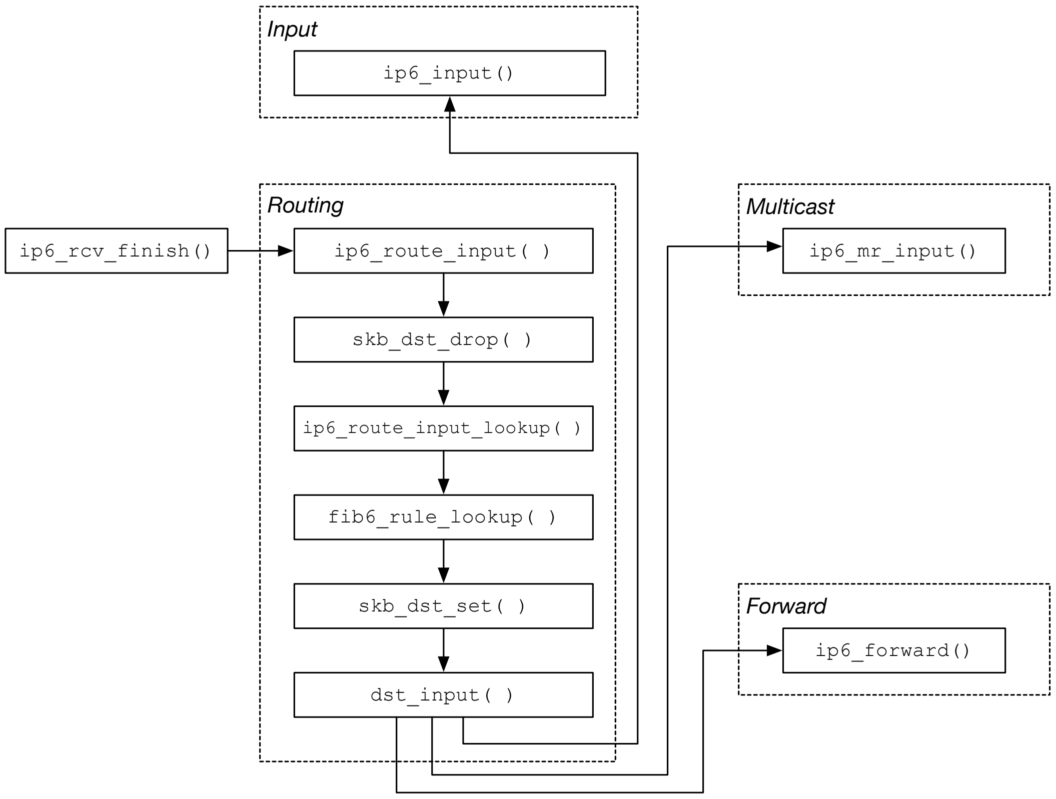


Figure A.3: Routing subsystem in Linux kernel

- If the kernel has PBR enabled and there are custom IPv6 FIB rules, then the fib_rules_lookup() is called to match the flow information (struct flowi6) with the custom defined rules.
- If the kernel has PBR enabled but there are no custom IPv6 FIB rules, then the ip6_pol_route_input() is called to match flow information (struct flowi6) with entries of the local FIB table.

- If the kernel has PBR disabled, then the same ip6_pol_route_input() is called to match flow information (struct flowi6) with entries of the main FIB table.

Once a match is found, the _skb_refdst of skb is set to point to dst (struct dst_entry) of the matched FIB entry (or FIB rule). Finally, the routing subsystem hands the packet to the next subsystem by calling the dst_input() function, which in turn invokes the input callback associated with the packet. As mentioned before, the input callback might be ip6_input(), ip6_forward(), lwtunnel_input(), or ip6_mc_input().

Input subsystem

The input subsystem handles packets destined to the local machine. The ip6_input() is called by the *Routing* subsystem which does nothing but invoke netfilter input (NF_INET_LOCAL_IN) registered callbacks, if any. IPv6 extension headers are parsed in the ip6_input_finish() via the appropriate handler for each extension header. For example, ipv6_rthdr_rcv() is the handler for IPv6 routing extension header (Next Header=43). The ip6_input_finish() function continues to parse received IPv6 packets until the transport layer protocol is found then the associated handler with such protocol is called (e.g., tcpv6, udpv6, icmpv6). In some cases, packets may be inserted again (re-cycled) into the routing subsystem (e.g. IPv6 packet with routing extension header).

Forward subsystem

If the routing decision was to forward a packet out, then the packet is passed to *Forwarding* subsystem by calling ip6_forward(). The *Forward* subsystem has to validate that IPv6 forwarding is enabled by checking the proc file system (procfs) configuration. Procfs is an interface that allows to configure kernel run-time parameters from user-space using the sysctl command [144]. IPv6 forwarding can be configured as shown in Listing A.4.

If the kernel is configured to forward IPv6 traffic, then ip6_forward() has to verify that the packet can be forwarded further ($hoplimit > 1$) otherwise it just drops the packet and sends back an ICMPv6 time exceeded message to the source of that packet [145]. It also has to validate that the packet size is less than or equal to path MTU. Then it decrements the hop limit of the packet and invokes the netfilter forward (NF_INET_FORWARD) registered callbacks,

Listing A.4: The Linux kernel configuration for IPv6 forwarding

```
# Enable IPv6 forwarding
$ sysctl -w net.ipv6.conf.all.forwarding=1
# Disable IPv6 forwarding
$ sysctl -w net.ipv6.conf.all.forwarding=0
```

if any. Eventually the ip6_forward_finish() passes the packet to the *Output* subsystem by invoking the output callback ip6_output() associated with the skb.

Multicast subsystem

The multicast subsystem processes packets with an IPv6 multicast destination address. However, processing of IPv6 multicast traffic is out of the scope of the thesis.

Local subsystem

Packets created on the local host are sent to the routing subsystem. If the packet destination address matches an entry in the the IPv6 routing table, then the packet will be handed to ip6_local_out(). It internally calls the __ip6_local_out() function to set the Payload Length field of the IPv6 header to be equal to the len field of skb. It also sets the protocol field value of skb to IPv6 (ETH_P_IPV6). Finally, netfilter (NF_INET_LOCAL_OUT) registered callbacks are invoked, if any, before handing the packet to the *Output* subsystem.

Output subsystem

Both forwarded and locally-generated packets are processed by the *Output* subsystem before being sent out. The ip6_output() function invokes the netfilter post-routing (NF_INET_POST_ROUTING) registered callbacks, if any, then passes the skb to the ip6_finish_output(), which in turn compares the packet size with the MTU. If fragmentation is needed, the ip6_fragment() function is called to handle it; otherwise, the ip6_finish_output2() function is called. The ip6_finish_output2() uses the *Neighbour* subsystem to get the layer-two (L2) address of the next hop.

Neighbor subsystem

The *Neighbor* subsystem is the packet's last stop right before the network driver. The $\mathrm{rt6\_nexthop()}$ function is used to get the L2 address of the next hop. The $\mathrm{\_\_ipv6\_neigh\_lookup\_noref()}$ searches into the neighbor table for the L2 address corresponding to the next hop. If there no entry for the next hop, the neighbour discovery process is invoked by calling the $\mathrm{\_neigh\_create()}$ function; otherwise, the $\mathrm{neigh\_output()}$ is invoked. Eventually the packet is pushed to the sending queue of the network device driver by calling $\mathrm{dev\_queue\_xmit()}$.

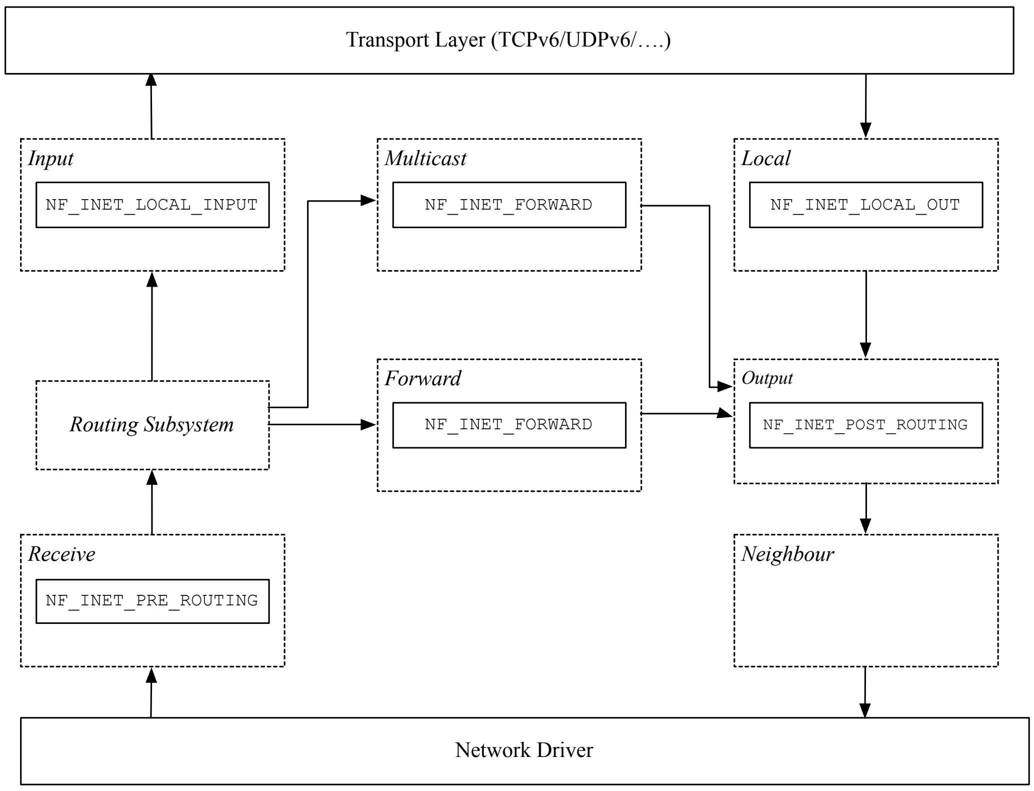


Figure A.4: Netfilter framework architecture

A.2 Netfilter framework

The netfilter framework is a set of hooks in the packet traversal path through the Linux network stack as shown in Figure A.4. It allows access to packets at different points [71]. The current netfilter implementation provides five different hooks distributed along the receive and transmit path of packets. The currently supported hooks can be described as follows:

- NF_INET_PRE_ROUTING: the first netfilter hook in the path of received packets. It allows to access received packets before being processed by the routing subsystem.
- NF_INET_INPUT: allows access to packets that are destined to the local machine. It is triggered after the routing subsystem decides to send the packet to local application.
- NF_INET_FORWARD: allows access to packets that have to be forwarded to another host. It is triggered after the routing subsystem decides to send the packet to another host.
- NF_INET_LOCAL_OUT: handles locally created outbound traffic.
- NF_INET_POST_ROUTING: triggered before handing the packets over to the network interface card.

Kernel modules can register callback functions at any of these hooks. A callback function can apply different kinds of processing to the packet. Packets can be modified or dropped by the callback function. Finally, the callback function returns the action to be taken on the packet to the netfilter hook. The following set of actions are currently supported by the netfilter hooks.

- NF_ACCEPT: instruct the kernel to continue the traversal of the packet normally.
- NF_DROP: stop the packet traversal though the network stack and just drop it.
- NF_STOLEN: notify the kernel that the callback will take care of the packet. The kernel simply has to forget about the packet.
- NF_QUEUE: instruct the kernel to push the packet into a specific queue (usually for user-space processing).
- NF_REPEAT: recall again all callback functions registered in the hook to process the packet one more time.

The netfilter provides a flexible way to add new features to the Linux kernel without the need to modify or recompile the kernel. The new features are implemented as kernel modules and then plugged into one of the netfilter hooks. The implementation of several network functions such as NAT and Firewall totally relies on netfilter framework.

A.3 SRv6 support

A.3.1 IPv6 routing extension headers processing

An IPv6 packet may carry zero, one, or more extension headers, each identified by the *Next Header* field of the preceding header. IPv6 extension headers (except hop-by-hop) are not examined or processed by any node along a packet's delivery path until the packet reaches the node identified in the destination address field of the IPv6 header [8].

In the Linux kernel, IPv6 extension headers are processed by the *Input* subsystem of the network stack. Except hop-by-hop, there is a protocol handler registered to process each different extension header. The registered protocol handler for all IPv6 routing extension headers types is ipv6_rthdr_rcv(). It is called from ip6_input_finish() once an IPv6 routing extension header is found while parsing an IPv6 packet. Firstly, ipv6_rthdr_rcv() checks the kernel configuration for processing source routing traffic. The Linux kernel configuration for accepting IPv6 source routing traffic (i.e. IPv6 packets that contains a Routing Header extension) is shown in Listing A.5.

Listing A.5: The Linux kernel configuration for IPv6 source routing

```
# Enable source routing generally
$ sysctl -w net.ipv6.conf.all.accept_source_route=1
# Enable source routing on a specific interface
$ sysctl -w net.ipv6.conf.<dev x>.accept_source_route=1
```

The default Linux kernel configuration is to drop all source routing packets. Note that this check is enforced when the IPv6 destination address of the packet is a local address, so that the received packet is processed by the *Input* subsystem.

- If the procfs configuration does not accept source routing (i.e., accept_source_route proc file is set to 0), the kernel drops the packet and replies to the sender with an ICMP parameter problem (ICMPV6_PARAMPROB) message, pointing to the routing header.
- If the procfs configuration allows source routing, then the processing is different for kernel before 4.10 or kernel 4.10 and later kernels as explained in Sections A.3.2 and A.3.3 respectively.

A.3.2 Kernels before 4.10

The source routing support before kernel 4.10 was mainly for routing header type-0, which was then deprecated. Given that the *procfs* configuration accepts source routing, the Linux kernel checks the Segments Left value and the processing will be as follows:

- If Segments Left is zero, the kernel ignores the routing header and proceeds to process the next header in the packet, whose type is identified by the Next Header field in the routing header.
- If Segments Left is non-zero, the kernel examines the routing Type field. If the type is unrecognized, the packet is dropped and an ICMP parameter problem message, pointing to the unrecognized Routing Type, is sent back to the packet's source address.

SRH extends the generic routing header defined by the IPv6 specifications, which already contains the Segments Left field. SRv6 traffic with Segments Left equal to zero is processed without any problem since the kernel does not look into the routing header. SRv6 traffic with Segments Left value greater than zero should be processed by the kernel. However, routing type 4 (SR) is not supported by kernels before 4.10. Accordingly, the kernel drops the packets and sends back an ICMP parameter problem message, pointing to the SRH type.

A.3.3 Kernel 4.10

The SRv6 capabilities were firstly merged in Linux kernel 4.10 [35]. The SRH is introduced and defined as struct ipv6_sr_hdr [146] as shown in Listing A.6.

SRv6 endpoint behaviors

The ipv6_srh_rcv() function [147] is added as a default handler for SRv6 traffic. It is called from ipv6_rthdr_rcv() for packets with routing type 4 (IPV6_SRCRT_TYPE_4). A per-interface sysctl (seg6_enabled) is added, to accept or deny received SRv6 traffic. If the sysctl is set, then SRH is processed as described in section 2.3.3, otherwise the packet is dropped silently. After SRH is processed and the packet IPv6 destination address is updated, the kernel feeds the packet again to the routing subsystem (ip6_route_input()) to be forwarded based on the new destination address.

Listing A.6: IPv6 segment routing header (SRH)

```
struct ipv6_sr_hdr {
    __u8                  nexthdr;
    __u8                  hdrlen;
    __u8                  type;
    __u8                  segments_left;
    __u8                  first_segment;
    __u8                  flags;
    __u16                 tag;
    struct in6_addr       segments[0];
};
```

The Linux kernel configuration for accepting SRv6 traffic is shown in Listing A.7. The seg6_enabled sysctl is configured by default to drop SRv6 traffic. All IPv6 addresses assigned to an SRv6-enabled interface are treated as local SIDs. In order to add a SID, one has to add an IPv6 address to an existing interface. The Linux kernel configuration for adding or deleting local SIDs is shown in Listing A.8.

Listing A.7: The Linux kernel configuration for SRv6 forwarding

```
# Enable receiving SRv6 generally
$ sysctl -w net.ipv6.conf.all.seg6_enabled=1
# Enable receiving SRv6 for a specific interface
$ sysctl -w net.ipv6.conf.<dev x>.seg6_enabled=1
```

Listing A.8: The Linux kernel configuration for configuring SRv6 SIDs

```
# Adding a new localsid
$ ip -6 addr add <IPv6 prefix> <dev x>
# Deleting an existing localsid
$ ip -6 addr del <IPv6 prefix> <dev x>
```

Listing A.9: Adding an SRv6 SID with transit behavior

```
$ ip -6 route add <prefix> encap seg6 mode \
  <encapmode> segs <segments> dev <device>
```

SRv6 transit behaviors

The SRv6 transit behaviors are implemented as Linux *lightweight* tunnels (seg6_iptunnel). A new encapsulation type (LWTUNNEL_ENCAP_SEG6) is added. Two new callbacks, seg6_input() and seg6_output(), are introduced to serve as input and output callbacks for seg6_iptunnel respectively. The ipv6_push_rthdr4() function was added to support SRH injection through setsockopt(), which enables the applications to define a per-socket SRH. The *iproute2* user-space utility is extended to support adding an SRv6 SID associated with an SRv6 transit behavior as shown in Listing A.9, where:

- prefix: IPv6 prefix of the route.
- encapmode: The SRv6 transit behavior. It can be encap to SRv6 *T.Encaps* behavior. It can be inline for SRv6 *T.Insert* behavior.
- segments: comma-separated list of segments (e.g., fc00::1,fc00::2).
- device: any non-loopback device.

An SRv6 SID with transit behavior is added as an IPv6 FIB entry into the kernel main routing table. It has seg6_input() as an input callback that calls seg6_do_srh() to perform the transit behavior corresponding to the matching SID. The transit could be *T.Insert* which is implemented in seg6_do_srh_inline(), or *T.Encaps* which is implemented in seg6_do_srh_encap(). After the SRH encapsulation is added to a received packet, the kernel feeds the packet again to the routing subsystem by calling ip6_route_input().

Limitations of SRv6 implementation in kernel 4.10

1. Any local IPv6 address is treated as an End SID. However, the network programming model states that SIDs should be explicitly enabled as such and associated with a specific endpoint behavior [34]. Hence, there is no way for the kernel to determine which endpoint behavior is associated with the active SID.
2. An IPv6 packet is processed by the SRv6 engine only if an SRH is present. However, the network programming model states that a node may receive a packet with an SRv6 SID in the destination address without an SRH. In such case the packet should still be processed by the SR engine.

A.3.4 kernel 4.14

Kernel 4.14 is another milestone for SRv6 support in Linux kernel. A new set of SRv6 behaviors has been added to the kernel [36]. A new Linux lightweight tunnel (seg6_local_lwtunnel) as well as a new encapsulation type (LWTUNNEL_ENCAP_SEG6_LOCAL) have been added to the kernel. The input and output callbacks for the new lightweight tunnel are seg6_local_input() and seg6_local_output() respectively. The netlink and iproute2 implementation were extended to support adding a localsid with the newly supported SRv6 behavior.

The supported SRv6 Endpoint behaviors included in kernel 4.14 are: *End.X*, *End.T*, *End.DX2*, *End.DX4*, *End.DX6*, *End.DT6*, *End.B6*, *End.B6.Encaps*. Along with the newly supported SRv6 Endpoint behaviors, some new transit behaviors have been added including: *T.Encaps4* which steers IPv4 packets through an SRv6 policy and *T.Encaps.L2* which encapsulates a packet along with the L2 frame (i.e. the received Ethernet header and its optional VLAN header) in the payload of the outer IPv6 packet. The seg6_iptunnel implementation has been extended with a new tunnel mode (SEG6_IPTUN_MODE_L2ENCAP) to support *T.Encaps.L2*. In order to support *T.Encaps4*, seg6_do_srh_encap() was slightly changed to accept the protocol family of the received packet.

Some generic SRv6 processing functions were added, that may be leveraged by other subsystems processing SRv6 packets including: get_srh() to parse IPv6 packets to get the SRH, decap_and_validate() to decapsulate and validate SRv6 packets, advance_nextseg() to advance to the next SID of the SRH and update the destination address of the IPv6 header with the next SID, and lookup_nexthop() to route the IPv6 packet after SRv6 behaviors have been applied. The SRv6 localsid table is supported. The kernel options CONFIG_IPV6_MULTIPLE_TABLES and CONFIG_IPV6_SUBTREES needs to be enabled. A tutorial of kernel 4.14 advanced configuration is provided in [148].

Limitations of SRv6 implementation in Linux kernel 4.14

1. Some SRv6 Endpoint behaviors are still not supported. For example, none of the SR proxy behaviors is supported [41].
2. The kernel option CONFIG_IPV6_MULTIPL_TABLES should be enabled, which, according to the IPv6 route lookup performance on Linux, has a fixed cost of 100 ns per lookup if the option is enabled [149].

Bibliography

[1] Y. Rekhter, S. Hares, and T. Li, “A Border Gateway Protocol 4 (BGP-4),” RFC 4271, Jan. 2006. [Online]. Available: https://rfc-editor.org/rfc/rfc4271.txt

[2] J. Moy, “OSPF Version 2,” RFC 2328, Apr. 1998. [Online]. Available: https://rfc-editor.org/rfc/rfc2328.txt

[3] “OSI IS-IS Intra-domain Routing Protocol,” RFC 1142, Feb. 1990. [Online]. Available: https://rfc-editor.org/rfc/rfc1142

[4] M. Shand and S. Bryant, “IP Fast Reroute Framework,” RFC 5714, Jan. 2010. [Online]. Available: https://rfc-editor.org/rfc/rfc5714.txt

[5] S. Previdi, C. Filsfils, B. Decraene, S. Litkowski, M. Horneffer, and R. Shakir, “Source Packet Routing in Networking (SPRING) Problem Statement and Requirements,” RFC 7855, May 2016. [Online]. Available: https://rfc-editor.org/rfc/rfc7855.txt

[6] K. R. Fall and W. R. Stevens, *TCP/IP Illustrated, Volume 1 : The Protocols, Second Edition*. Addison-Wesley Professional Computing Series, Dec 2011, vol. 1.

[7] “Internet Protocol,” RFC 791, Sep. 1981. [Online]. Available: https://rfc-editor.org/rfc/rfc791.txt

[8] D. S. E. Deering and B. Hinden, “Internet Protocol, Version 6 (IPv6) Specification,” RFC 8200, Jul. 2017. [Online]. Available: https://rfc-editor.org/rfc/rfc8200.txt

[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, Jan 2015.

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: Enabling innovation in campus networks,” *SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[11] 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, Feb 2015.

[12] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[13] J. M. Halpern and C. Pignataro, "Service Function Chaining (SFC) Architecture," RFC 7665, Oct. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7665.txt

[14] P. Quinn and T. Nadeau, "Problem Statement for Service Function Chaining," RFC 7498, Apr. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7498.txt

[15] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, "Segment Routing Architecture," RFC 8402, Jul. 2018. [Online]. Available: https://rfc-editor.org/rfc/rfc8402.txt

[16] C. Blogs, "Celebrating Segment Routing's 5th Birthday," https://blogs.cisco.com/sp/celebrating-segment-routings-5th-birthday.

[17] "SRv6 Scientific Papers and Research Publications," http://www.segment-routing.net/scientific-papers/.

[18] A. Viswanathan, E. C. Rosen, and R. Callon, "Multiprotocol Label Switching Architecture," RFC 3031, Jan. 2001. [Online]. Available: https://rfc-editor.org/rfc/rfc3031.txt

[19] F. Gont, R. Atkinson, and C. Pignataro, "Recommendations on Filtering of IPv4 Packets Containing IPv4 Options," RFC 7126, Feb. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7126.txt

[20] G. Neville-Neil, P. Savola, and J. Abley, "Deprecation of Type 0 Routing Headers in IPv6," RFC 5095, Dec. 2007. [Online]. Available: https://rfc-editor.org/rfc/rfc5095.txt

[21] "What is VPP ?" https://wiki.fd.io/view/VPP.

[22] S. Matsushima, C. Filsfils, Z. Ali, Z. Li, and K. Rajaraman, "SRv6 Implementation and Deployment Status," Internet Engineering Task Force, Internet-Draft draft-matsushima-spring-srv6-deployment-status-07, Apr. 2020,

work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-matsushima-spring-srv6-deployment-status-07

[23] C. Filsfils, F. Clad, P. C. Garvia, A. Abdelsalam, S. Salsano, O. Bonaventure, J. Horn, and J. Liste, "SRv6 interoperability report," Internet Engineering Task Force, Internet-Draft draft-filsfils-spring-srv6-interop-02, Mar. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-filsfils-spring-srv6-interop-02

[24] A. Bashandy, C. Filsfils, S. Previdi, B. Decraene, S. Litkowski, and R. Shakir, "Segment Routing with the MPLS Data Plane," RFC 8660, Dec. 2019. [Online]. Available: https://rfc-editor.org/rfc/rfc8660.txt

[25] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois, "The Segment Routing architecture," in *Global Communications Conference (GLOBECOM), 2015 IEEE.* IEEE, 2015, pp. 1–6.

[26] R. Enns, M. Björklund, A. Bierman, and J. Schönwälder, "Network Configuration Protocol (NETCONF)," RFC 6241, Jun. 2011. [Online]. Available: https://rfc-editor.org/rfc/rfc6241.txt

[27] J. Vasseur and J.-L. L. Roux, "Path Computation Element (PCE) Communication Protocol (PCEP)," RFC 5440, Mar. 2009. [Online]. Available: https://rfc-editor.org/rfc/rfc5440.txt

[28] "OpenDaylight Project," https://www.opendaylight.org.

[29] "Internet Assigned Numbers Authority," https://www.iana.org/.

[30] D. B. Johnson, J. Arkko, and C. E. Perkins, "Mobility Support in IPv6," RFC 6275, Jul. 2011. [Online]. Available: https://rfc-editor.org/rfc/rfc6275.txt

[31] 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://rfc-editor.org/rfc/rfc8754.txt

[32] IANA, "Protocol Numbers," https://www.iana.org/assignments/protocol-numbers/protocol-numbers.txt.

[33] Wikipedia, "Type-length-value," https://en.wikipedia.org/wiki/Type-length-value.

[34] C. Filsfils, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima, and Z. Li, “SRv6 Network Programming,” Internet Engineering Task Force, Internet-Draft draft-ietf-spring-srv6-network-programming-16, Jun. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-spring-srv6-network-programming-16

[35] D. Lebrun and O. Bonaventure, “Implementing IPv6 Segment Routing in the Linux Kernel,” in *Proceedings of the Applied Networking Research Workshop*. ACM, 2017, pp. 35–41.

[36] D. Lebrun, “Reaping the benefits of ipv6 segment routing,” 2017. [Online]. Available: https://inl.info.ucl.ac.be/system/files/phdthesis-lebrun.pdf

[37] A. Abdelsalam *et al.*, “Implementation of Virtual Network Function chaining through Segment Routing in a Linux-based NFV infrastructure,” in *2017 IEEE Conference on Network Softwarization (NetSoft)*, July 2017, pp. 1–5.

[38] 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 *5th IEEE International Conference on Network Softwarization (NetSoft 2019)*. IEEE, June 2019.

[39] P. L. Ventre, S. Salsano, M. Polverini, A. Cianfrani, A. Abdelsalam, C. Filsfils, P. Camarillo, and F. Clad, “Segment Routing: a Comprehensive Survey of Research Activities, Standardization Efforts and Implementation Results,” *arXiv preprint arXiv:1904.03471v2*, 2019.

[40] A. Abdelsalam, “Demo: Chaining of Segment Routing aware and unaware Service Functions,” in *IFIP Networking 2018*. IEEE, May 2018.

[41] F. Clad, X. Xu, C. Filsfils, daniel.bernier@bell.ca, 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-02, Mar. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-spring-sr-service-programming-02

[42] B. Thekkedath, *Network Functions Virtualization For Dummies*. Wiley, 2016.

[43] T. Li, D. Farinacci, S. P. Hanks, D. Meyer, and P. S. Traina, “Generic Routing Encapsulation (GRE),” RFC 2784, Mar. 2000. [Online]. Available: https://rfc-editor.org/rfc/rfc2784.txt

[44] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks," RFC 7348, Aug. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7348.txt

[45] H.-W. Braun, "Models of policy based routing," RFC 1104, Jun. 1989. [Online]. Available: https://rfc-editor.org/rfc/rfc1104.txt

[46] 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*, May 2018, pp. 46–54.

[47] "DPDK." [Online]. Available: https://www.dpdk.org/

[48] "srext - a Linux kernel module implementing SRv6 Network Programming model." [Online]. Available: https://github.com/netgroup/SRv6-net-prog/

[49] "VirtualBox home page," http://www.virtualbox.org/.

[50] "Vagrant home page," http://www.vagrantup.com/.

[51] "SRv6 open source softwares," https://www.segment-routing.net/open-software/.

[52] Kentaro Ebisawa, "JANOG43 Forefront of SRv6, Open Source Implementations," https://www.slideshare.net/kentaroebisawa/janog43-forefront-of-srv6-open-source-implementations.

[53] "SRNK- An SR proxy for SR-unaware VNFs." [Online]. Available: https://netgroup.github.io/srnk/

[54] "SRNK - kernel implementation." [Online]. Available: https://github.com/netgroup/srnk-kernel

[55] "SRNK - iproute2 implementation." [Online]. Available: https://github.com/netgroup/srnk-iproute2

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

[57] "Open vSwitch." [Online]. Available: http://openvswitch.org

[58] "OpenvSwitch nsh patch." [Online]. Available: https://github.com/pritesh/ovs/tree/nsh-v8

[59] OpenDaylight Project, “Service function chaining,” http://wiki.opendaylight.org/view/Service_Function_Chaining:Main.

[60] “Neutron home page,” https://wiki.openstack.org/wiki/Neutron.

[61] “OpenStack home page,” https://www.openstack.org/.

[62] “Neutron - Service Function Chaining,” http://wiki.openstack.org/wiki/Neutron/ServiceInsertionAndChaining.

[63] A. Abdelsalam, S. Salsano, F. Clad, P. Camarillo, and C. Filsfils, “SR-Snort: IPv6 Segment Routing Aware IDS/IPS,” in *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov 2018, pp. 1–2.

[64] “Snort- network intrusion detection & prevention system.” [Online]. Available: https://www.snort.org

[65] “The netfilter.org "nftables" project.” [Online]. Available: https://netfilter.org/projects/nftables/

[66] “Tcpdump/libpcap public repository.” [Online]. Available: https://www.tcpdump.org

[67] J. R. Vacca and S. Ellis, *Firewalls: Jump start for Network and Systems Administrators*. Elsevier, 2005.

[68] “SERA - SEgment Routing Aware Firewall,” https://github.com/SRouting/SERA.

[69] KernelNewbies, “Linux 4.16 changelog,” https://kernelnewbies.org/Linux_4.16.

[70] “iptables releases. iptables-1.6.2 changelog,” https://netfilter.org/projects/iptables/downloads.html#iptables-1.6.2, February 2018.

[71] R. Russell and H. Welte, “Linux netfilter Hacking Howto,” http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.html.

[72] “CloudLab home page,” https://www.cloudlab.us/.

[73] Robert Ricci, Eric Eide, and the CloudLab Team, “Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications,” *; login:: the magazine of USENIX & SAGE*, vol. 39, no. 6, pp. 36–38, 2014.

[74] “Kernel 4.14 release,” https://kernelnewbies.org/Linux_4.14.

[75] "Linux Foundation Wiki - iproute2." [Online]. Available: https://wiki.linuxfoundation.org/networking/iproute2

[76] "iPerf - The ultimate speed test tool for TCP, UDP and SCTP." [Online]. Available: http://iperf.fr

[77] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks." in *Lisa*, vol. 99, no. 1, 1999, pp. 229–238.

[78] "The libpcap interface to various kernel packet capture mechanism." [Online]. Available: https://github.com/the-tcpdump-group/libpcap

[79] "Snort Users Manual." [Online]. Available: http://manual-snort-org.s3-website-us-east-1.amazonaws.com/

[80] "SR-Snort: IPv6 Segment Routing Aware Snort." [Online]. Available: https://github.com/SRouting/SR-Snort

[81] A. Abdelsalam, S. Salsano, F. Clad, P. Camarillo, and C. Filsfils, "SR-Snort: IPv6 Segment Routing Aware IDS/IPS," in *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov 2018, pp. 1–2.

[82] "nftables 0.8.4 release." [Online]. Available: https://lwn.net/Articles/753302/

[83] "Tcpdump- Add support for IPv6 routing header type 4." [Online]. Available: https://github.com/the-tcpdump-group/tcpdump/commit/9c33608cb2fb6a64e1b76745efa530a63de08100

[84] "Tcpdump - Fix checksum calculation for IPv6 Segment Routing (SRv6) traffic." [Online]. Available: https://github.com/the-tcpdump-group/tcpdump/commit/a87d6a044893dace0534e91d77ce236a101d5794

[85] M. Konstantynowicz *et al.*, "Benchmarking and Analysis of Software Data Planes," Dec 2017. [Online]. Available: https://fd.io/wp-content/uploads/sites/34/2018/01/performance_analysis_sw_data_planes_dec21_2017.pdf

[86] S. Bradner and J. McQuaid, "Benchmarking Methodology for Network Interconnect Devices," Internet Requests for Comments, RFC Editor, RFC 2544, March 1999. [Online]. Available: https://tools.ietf.org/html/rfc2544

[87] A. Abdelsalam, P. L. Ventre, A. Mayer, S. Salsano, P. Camarillo, F. Clad, and C. Filsfils, "Performance of ipv6 segment routing in linux kernel," in *2018 14th International Conference on Network and Service Management (CNSM)*, Nov 2018, pp. 414–419.

[88] A. Abdelsalam, P. L. Ventre, C. Scarpitta, A. Mayer, S. Salsano, P. Camarillo, F. Clad, and C. Filsfils, "Srperf: a performance evaluation framework for ipv6 segment routing," *arXiv preprint arXiv:2001.06182*, 2020.

[89] "TRex realistic traffic generator." [Online]. Available: https://trex-tgn.cisco.com/

[90] "Trex stateless python api." [Online]. Available: https://trex-tgn.cisco.com/trex/doc/cp_stl_docs/index.html

[91] "Scapy - packet crafting for python," https://scapy.net/.

[92] O. Ben-Kiki *et al.*, "Yaml ain't markup language," http://www.yaml.org/spec/1.2/spec.html, 2009.

[93] "PyYAML." [Online]. Available: https://pyyaml.org

[94] "JSON-RPC." [Online]. Available: https://www.jsonrpc.org

[95] "ZeroMQ." [Online]. Available: https://zeromq.org

[96] "SRPerf - Performance Evaluation Framework for Segment Routing." [Online]. Available: https://github.com/SRouting/SRPerf

[97] S. Bradner, "Benchmarking Terminology for Network Interconnection Devices," Internet Requests for Comments, RFC Editor, RFC 1242, July 1991. [Online]. Available: https://tools.ietf.org/html/rfc1242

[98] "CSIT REPORT - The Fast Data I/O Project (FD.io) Continuous System Integration and Testing (CSIT) project report for CSIT master system testing of VPP-18.04 release." [Online]. Available: https://docs.fd.io/csit/master/report/_static/archive/csit_master.pdf

[99] A. Hothan *et al.*, "NFVBench Documentation - Release 1.5.1," June 2018. [Online]. Available: https://media.readthedocs.org/pdf/opnfv-nfvbench/stable/opnfv-nfvbench.pdf

[100] "ethtool - Linux man page." [Online]. Available: https://linux.die.net/man/8/ethtool

[101] A. Shaw, "Multi-queue network interfaces with SMP on Linux," https://greenhost.nl/2013/04/10/multi-queue-network-interfaces-with-smp-on-linux/.

[102] T. Szigeti, D. Zacks, M. Falkner, and S. Arena, *Cisco Digital Network Architecture: Intent-based Networking for the Enterprise*. Addison-Wesley Professional Computing Series, Dec 2011, vol. 1.

[103] L. kernel documentations, “Scaling in the Linux Networking Stack,” https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[104] ——, “SMP IRQ affinity,” https://www.kernel.org/doc/Documentation/IRQ-affinity.txt.

[105] J. Corbet, “Large receive offload,” https://lwn.net/Articles/243949/.

[106] ——, “Generic receive offload,” https://lwn.net/Articles/358910/.

[107] H. X. Corbet, “GSO: Generic Segmentation Offload,” https://lwn.net/Articles/188489/.

[108] “SRPerf-measurements - The performance measurements of the SRv6 behaviors.” [Online]. Available: https://github.com/SRouting/SRPerf-measurements

[109] “Linux-SRPerf - Linux kernel for SRPerf,” https://github.com/SRouting/Linux-SRPerf.

[110] S. Seth and M. A. Venkatesulu, *Transmission and Reception of Packets*. John Wiley & Sons, Ltd, 2008, ch. 18, pp. 697–722. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470377833.ch18

[111] L. Rizzo, “netmap: A novel framework for fast packet i/o,” in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, 2012, pp. 101–112. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo

[112] “DPDK - Supported Hardware.” [Online]. Available: https://core.dpdk.org/supported/

[113] “Network Function Framework for Go.” [Online]. Available: https://github.com/intel-go/nff-go

[114] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, D. Rossi, and J. Tollet, “Batched packet processing for high-speed software data plane functions,” in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, April 2018, pp. 1–2.

[115] T. Begin, B. Baynat, G. A. Gallardo, and V. Jardin, “An accurate and efficient modeling framework for the performance evaluation of dpdk-based virtual switches,” *IEEE Transactions on Network and Service Management*, 2018.

[116] N. Pitaev, M. Falkner, A. Leivadeasy, and I. Lambadarisy, "Multi-vnf performance characterization for virtualized network functions," in *2017 IEEE Conference on Network Softwarization (NetSoft)*, July 2017, pp. 1–5.

[117] N. Pitaev, M. Falkner, A. Leivadeas, and I. Lambadaris, "Characterizing the performance of concurrent virtualized network functions with ovs-dpdk, fd. io vpp and sr-iov," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 2018, pp. 285–292.

[118] "Open vSwitch with DPDK." [Online]. Available: http://docs.openvswitch.org/en/latest/intro/install/dpdk/

[119] "Open Platform for NFV (OPNFV)." [Online]. Available: https://www.opnfv.org/

[120] "NFVbench home." [Online]. Available: https://wiki.opnfv.org/display/nfvbench/NFVbench

[121] "VSPERF home," https://wiki.opnfv.org/display/vsperf/VSperf+Home.

[122] "VSPERF CI Results." [Online]. Available: https://wiki.opnfv.org/display/vsperf/VSPERF+CI+Results

[123] Maciek Konstantynowic and Patrick Lu and Shrikant M. Shah, "Benchmarking and Analysis of Software Data Planes," https://fd.io/wp-content/uploads/sites/34/2018/01/performance_analysis_sw_data_planes_dec21_2017.pdf.

[124] "Linux is the largest software development project on the planet," https://www.cio.com/article/3069529/linux/linux-is-the-largest-software-development-project-on-the-planet-greg-kroah-hartman.html.

[125] Wikipedia, "History of Linux," https://en.wikipedia.org/wiki/History_of_Linux.

[126] G. Xiao, Z. Zheng, and H. Wang, "Evolution of linux operating system network," *Physica A: Statistical Mechanics and its Applications*, vol. 466, pp. 249–258, 2017.

[127] A. Israeli and D. G. Feitelson, "The linux kernel as a case study in software evolution," *Journal of Systems and Software*, vol. 83, no. 3, pp. 485–501, 2010.

[128] J. Corbet and G. Kroah-Hartman, "2017 Linux Kernel Development Report," https://www.linuxfoundation.org/2017-linux-kernel-report-landing-page/.

[129] “The Linux Foundation,” https://www.linuxfoundation.org/about/.

[130] “The premier news and information source for the free software community,” https://lwn.net/.

[131] “Linux Kernel Documentation,” https://www.kernel.org/doc/.

[132] “The Linux man-pages project,” https://www.kernel.org/doc/man-pages/.

[133] “The kernel mailing list archives,” https://lkml.org/.

[134] “Socket buffer header file,” https://elixir.free-electrons.com/linux/latest/source/include/linux/skbuff.h.

[135] G. M. Trotter, “Terminology for Forwarding Information Base (FIB) based Router Performance,” RFC 3222, Dec. 2001. [Online]. Available: https://rfc-editor.org/rfc/rfc3222.txt

[136] “IPv6 FIB header file,” https://elixir.free-electrons.com/linux/latest/source/include/net/ip6_fib.h.

[137] “Rtnetlink "rtnetlink.h",” http://elixir.free-electrons.com/linux/latest/source/include/uapi/linux/rtnetlink.h.

[138] “Protocol independent destination cache definitions,” https://elixir.free-electrons.com/linux/latest/source/include/net/dst.h.

[139] A. Kleen, H. M. Khosravi, A. Kuznetsov, and J. H. Salim, “Linux Netlink as an IP Services Protocol,” RFC 3549, Jul. 2003. [Online]. Available: https://rfc-editor.org/rfc/rfc3549.txt

[140] “Linux Kernel 4.3,” https://kernelnewbies.org/Linux_4.3.

[141] D. S. E. Deering and B. Hinden, “IP Version 6 Addressing Architecture,” RFC 4291, Feb. 2006. [Online]. Available: https://rfc-editor.org/rfc/rfc4291.txt

[142] “IPv6 Routing Policy Rules implementation,” http://elixir.free-electrons.com/linux/latest/source/net/ipv6/fib6_rules.c.

[143] “IPv6 Forwarding Information Database ’fib6’ implementation,” http://elixir.free-electrons.com/linux/latest/source/net/ipv6/ip6_fib.c.

[144] “The proc file system,” https://www.kernel.org/doc/Documentation/filesystems/proc.txt.

[145] M. Gupta and A. Conta, “Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification,” RFC 4443, Mar. 2006. [Online]. Available: https://rfc-editor.org/rfc/rfc4443.txt

[146] “SR-IPv6 implementation ’seg6’,” https://elixir.free-electrons.com/linux/latest/source/include/uapi/linux/seg6.h.

[147] “Extension Header handling for IPv6,” http://elixir.free-electrons.com/linux/latest/source/net/ipv6/exthdrs.c.

[148] “SRv6 - Advanced Configuration,” http://segment-routing.org/index.php/Implementation/AdvancedConf.

[149] V. Bernat, “IPv6 route lookup on Linux,” https://vincent.bernat.im/en/blog/2017-ipv6-route-lookup-linux.