UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

An interface for programmable IPv6 Segment Routing network functions in Linux

Dissertation presented by
Mathieu XHONNEUX

for obtaining the Master's degree in
Electrical Engineering

Supervisor(s)
Olivier BONAVENTURE

Reader(s)
Olivier TILMANS, David BOL

Academic year 2017-2018

Abstract

In modern networks, operators often deploy intermediate *network functions* capable of analyzing and altering traffic. IP networks have been built upon the key principle that routers must forward packets along the shortest path between source and destination. This behavior is not always adapted to the deployment of network functions. A recent network architecture, *Segment Routing*, allows traffic flows to be steered across specific paths and opens up new possibilities for the development of network functions. This architecture operates at the network layer, as a consequence, network functions leveraging Segment Routing must be developed in the kernel of the operating system, network layer mechanisms being usually implemented there. However, developing features at the kernel level is a cumbersome process. The purpose of this work is to study how the development of Segment Routing network functions can be facilitated. To this end, we implement an interface for programmable IPv6 Segment Routing (SRv6) functions in Linux. This interface leverages BPF, an in-kernel virtual machine allowing to inject at runtime user-specific code in kernel components. SRv6 network functions can be developed as independent C programs and dynamically loaded into the network stack using this interface. They can then leverage the functionalities of the SRv6 data plane. The flexibility and performance of this interface is assessed through the development of three use-cases, implementing OAM and load balancing services. The implementation of these use-cases demonstrate that the SRv6 BPF interface provides the required flexibility to implement a wide range of network functions. Experimental results show that the use of BPF induces almost no performance overhead. Finally, this feature has been sent upstream and will be released in Linux 4.18.

Acknowledgments

I would like to express my sincere gratitude to Prof. Olivier Bonaventure for his passionate and insightful guidance. I would also like to thank Olivier Tilmans, Fabien Duchêne and Mathieu Jadin for their suggestions, assistance, and patience despite my frequent and unannounced calls for help in their offices.

Talk is cheap. Show me the code.

— Linus Torvalds

https://github.com/Zashas/Thesis-SRv6-BPF

Contents

1 Introduction

Since the inception of the ARPANET, computer networks have merely been perceived as systems allowing to transfer digital information from a device to another. However, this understanding does not hold for most modern networks. Operators often deploy intermediate *network functions* capable of analyzing and altering traffic. The usage of such functions spreads over a wide range of services, such as firewalls, load balancers, monitoring tools, content caches, etc. In the end, networks can no longer be assimilated to simple pipes between devices, but are full fledged computing systems.

Nevertheless, the use of network functions was not foreseen when the foundations of computer networking were designed. IP networks have been built upon the key principle that routers must forward packets along the shortest path between source and destination. This behavior is not always adapted to the deployment of intermediate network functions, as they often require traffic flows to follow alternative paths. Implementing such mechanisms in traditional networks often requires heavy topological or configuration modifications.

In the attempt to push the limits of classic IP networks, *Segment Routing*, a new source-routing architecture has been proposed [1]. Segment Routing allows source nodes to efficiently steer traffic flows across specific paths in the network. To do so, nodes can inject in the packets they send an ordered list of routers which will be used as path to reach the destination. As a consequence, this architecture enables the design of a broad new range of network functions that were previously impossible or difficult to implement.

Yet, even if Segment Routing theoretically opens up new possibilities, practical issues need to be resolved first. This architecture operates at the network layer, hence new functions relying on it should be implemented close to this layer. However, in most operating systems, the network layer mechanisms take place inside their kernels. This is a major obstacle, since implementing new Segment Routing features hence implies to modify the operating system itself. This is impossible if the operating system is closed source, and even in the case of open source systems, e.g. Linux, implementing new features at the kernel level is usually a difficult and time-consuming task.

Therefore, in order to truly leverage the theoretical possibilities of Segment Routing, the following interrogation has to be answered first: is it possible to design a suitable interface for the development of network functions that leverage the Segment Routing architecture ? In other words, could the forwarding behaviors of a major operating system be easily and efficiently extended without having to modify the system ?

This work addresses this question in the context of the IPv6 flavor of Segment Routing (SRv6). Unlike previous work, our objective is to implement a programming interface for network functions that does not bypass the kernel. The open source operating system Linux is used as candidate for our implementation.

Structure of the thesis

This master thesis is organized as follows:

- In **Chapter 2**, the Segment Routing architecture and its IPv6 declination are thoroughly described. We notably insist on the inherent properties of Segment Routing that make it particularly relevant for the development of network functions.
- **Chapter 3** introduces BPF, a virtual machine running inside the Linux kernel. This recent feature allows to add at runtime user-specific behaviors in specific datapaths of the kernel. Concise C programs can be injected in the kernel, provided there is an adapted interface to the mechanism that needs to be tweaked.
- Leveraging the SRv6 implementation of Linux and BPF, **chapter 4** retraces how we developed a BPF interface enabling network operators to efficiently implement Segment Routing network functions. The performance of this interface is then assessed via micro-benchmarks.
- **Chapter 5** describes the set of tools that we developed in the frame of this work. This set includes a unit testing framework for SRv6, a library for SRv6 BPF network functions, and a user space daemon infrastructure capable of interacting with network functions.
- Finally, we developed three use-cases relying on our SRv6 BPF interface: a passive one-way delay monitoring system, a solution for aggregating multiple access networks and a deterministic traceroute tool for multipaths networks. **Chapter 6** describes, for each use-case, the architecture of the solution, how it has been implemented in BPF, and how it performed in experimental evaluations.

Finally, all the code written for this work has been released under GNU General Public License v3.0, and is available at `https://github.com/Zashas/Thesis-SRv6-BPF`.

2 Segment Routing

Nowadays, a great majority of computer networks rely on hop-by-hop routing to forward packets between endhosts. In this paradigm, nodes perform destination-based forwarding and send traffic along the shortest path between the source and the destination. The Internet Protocol (IP), in its versions 4 and 6, is the most widespread implementation of hop-by-hop routing.

In IP networks, each node, router or endhost, is assigned a unique IP address, which identifies the node inside the network, and contains a routing table. Routing tables are a set of routes, i.e. mappings between IP addresses and nexthops. Nexthops are other nodes directly connected to the current one. Each route contains the nexthop that is the closest to the associated destination. IP nodes achieve hop-by-hop routing by performing look-ups in their Forwarding Information Base (FIB)[1] to retrieve the nexthop associated to the destination, and then forward the packet to this nexthop.

Although IP networks are widely deployed, they also exhibit some drawbacks. In particular, network operators may need to define network policies that go against the hop-by-hop routing paradigm. For example, real-time applications often require their traffic to be forwarded over low-latencies paths, however the shortest path does not necessarily have the lowest latency. This situation is an instance of *traffic engineering* problems. Generally speaking, pure IP networks do not are not capable of efficiently handling such issues. Multiprotocol Label Switching (MPLS) [2], along with the RSVP-TE protocol [3], have been designed with TE in mind, and are the standard solutions to implement TE policies. Yet, MPLS exhibits severe scaling limitations [4], and the Segment Routing (SR) architecture has been proposed as an alternative in 2015 [1].

Since then, SR has sparked a growing interest in academic research. Its architecture not only allows to implement TE policies [5], but can also be leveraged in Software-Defined Networks (SDN) [6] and for Service Function Chaining (SFC) [7, 8]. Even if this architecture has originally been designed for the MPLS dataplane, it has been adapted to IPv6 as well [9].

This chapter summarizes how Segment Routing leverages the source routing paradigm. Its IPv6 declination is subsequently explained. Finally, the SRv6 implementation in the Linux kernel is described.

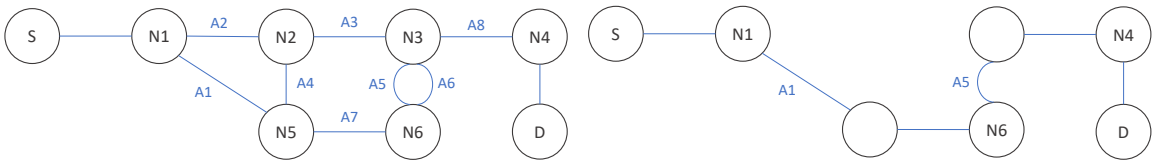
2.1 A source routing architecture

SR relies on the source routing paradigm, where the source, an endhost or edge router, may decide over which paths packets must be forwarded [10]. These paths are not necessarily the shortest ones, and can be chosen according to any arbitrary criteria.

1. A routing table may contain several routes towards a same destination. In such situations, the network operator can define a preference order between routes by defining their *metric*. The FIB is the subset of all routes being actually used to forward packets.

Each path specified by the source is decomposed into an ordered list of instructions, called *segments*. Each segment represents a function to be executed at a specific location in the network. These functions may range from simple topological instructions (e.g. forwarding a packet on a specific link) to more complex user-defined behaviors. *Network programming* [1] is defined as the combination of SR functions, with the goal to achieve networking or application-specific objectives that are not limited to simple packet forwarding.

[10] defines the two following principal topological instructions. A node segment (or *Node-SID*[2]) forces the packet to go through a particular node in the network. An adjacency segment (or *Adj-SID*) steers the packet through a specific link. These two instructions are illustrated in figure 2.1.



(a) Example of network topology with Node-SIDs and Adj-SIDs.

(b) Example of SR forwarding for a packet with the segment list $< N_1, A_{1/1\to2}, N_6, A_{5/6\to3}, N_4, D >$.

Figure 2.1: Illustration of Segment Routing topological instructions.

The segment list, representing the specified path to follow, is encoded within the packet itself. Unlike hop-by-hop routing, SR routers can be stateless. The whole state required for the forwarding of each packet is contained in the segment list. A cursor in the packet indicates which segment is being currently processed. Once the packet arrives to the router advertising the segment, this router advances the cursor and executes the instruction associated to the segment. This process is repeated until the final destination is reached.

2.2 IPv6 Segment Routing

Segment Routing in the IPv6 data plane is implemented by adding to the IPv6 header an extension header called the Segment Routing Header (SRH) [9]. In addition of this extension header, additional segment types specific to SRv6 have been defined in [8]. We now summarize these two IETF drafts.

2.2.1 The Segment Routing Header

The SRH is an instance of the Routing Header, an IPv6 extension header that predates SR [11].[3] The structure of the SRH is presented in figure A.1, its fields are now described.[4]

In SRv6, segments are encoded as regular 128-bit IPv6 addresses. The segment list is contained in

2. SID stands for *Segment Identifier*. They may be used interchangeably.

3. The initial IPv6 specifications defined the Routing Header of type 0 (RH0), but this header has since been deprecated due to security reasons.

4. These descriptions are based on the 14th revision of the IETF Segment Routing Header draft [9], which is the latest available at the time of writing.

the SRH, up to 128 segments can be inserted. Segments are stored in reverse order in the segment list, i.e. the segment at index 0 corresponds to the last instruction to execute.

The four bytes of `Next Header` indicate which type of header is following the SRH.[5] `Hdr Ext Len` encodes the length of the segments and possible TLVs as an 8-byte multiple. `Routing Type` is always set to 4. `Segments Left` (SL) indicates the number of remaining segments. It acts as an index of the segment list, the entry corresponding to this index represents the current segment. `Last Entry` specifies the index of the first segment in the segment list. Concerning `Flags`, a single flag is defined, the H-flag, indicating if an HMAC TLV is present. `Tag` can be used to mark the packet as belonging to a class of packets sharing the same properties.

Finally, Type-Length-Value (TLV) fields can be inserted in the bottom part of the SRH. TLVs are 3-tuples that can be used to store additional data in the SRH. The structure of this 3-tuple allows TLVs of variable length to be implemented, the `Length` field specifies the number of bytes contained in `Value`. `Type` stores a unique identifier for each TLV type. [9] defines the 3 following TLVs:

- HMAC TLV: contains a Keyed-Hash Message Authentication Code (HMAC). The HMAC TLV is used to authenticate the SRH. This security mechanism is needed to prevent denial of service attacks, where an attacker forges a SRH oscillating between two nodes. This TLV is optional. If it is appended to the SRH, the H-flag must be set.
- PAD0 TLV: a single byte of padding (figure A.2).
- PADN TLV: between 2 and 7 bytes of padding (figure A.3).

Besides these three structures, other TLVs may be defined if needed. The network functions implemented in chapter 6 rely on TLVs for Operations Administration and Maintenance (OAM) features, that are not cited in the above list. Since the `Hdr Ext Len` field is encoded as an 8-byte multiple, the size in bytes of the whole SRH must be divisible by 8. If this condition is not respected, a single padding TLV (PAD0 or PADN) may be inserted at the end of the SRH.

2.2.1.1 Processing of SRv6 packets

With the introduction of the SRH, new packet processing rules for SRv6 packets have to be defined. We first classify the three types of nodes that can be involved in SRv6 packet processing:

- Source SR Node: a node inserting a SRH into an IPv6 packet. This can either be an endhost originating an IPv6 packet, or a SRv6 ingress router encapsulating a received packet in an outer IPv6 header with a SRH.
- Transit Node: a node forwarding an IPv6 packet whose destination address (DA) is not a SID belonging to the node.
- SR Segment Endpoint Node: a node receiving an IPv6 packet whose DA is a SID belonging

5. The values of the `Next Header` field of the IPv6 header are re-used here, e.g. 6 for TCP, 17 for UDP, 41 for IPv6 (IPv6 in IPv6 encapsulation) [11].

to the node.

The forwarding rules are different depending on the type of SR node. Source SR nodes first set the DA of the IPv6 header to the first segment of the SRH, allowing the regular IPv6 forwarding to be performed. Transit nodes operate as pure IPv6 routers, i.e. they do not read the SRH, and only forward the packet based on its DA. Hence, if two SR Segment Endpoint SIDs follow each other in a SRH, and if their corresponding physical nodes are not directly connected, hop-by-hop shortest path routing is carried out between these nodes.

Finally, SR Segment Endpoint nodes follow the algorithm presented in listing 1.

Listing 1 SRH processing for endpoint nodes.

```
1  IF DA = local SID (segment endpoint)
2      IF Segments Left > 0 then
3          Decrement Segments Left
4          Update DA with Segment List[Segments Left]
5          FIB lookup on the updated DA
6          Forward the packet out based on the FIB lookup
7      ELSE
8          Continue regular IPv6 processing of the packet (e.g. decapsulation and
               forward)
9          End of processing
10     END IF
11 END IF
```

2.2.2 SRv6 network programming

The forwarding rules introduced above are only capable of implementing the Node-SID instruction from the generic SR architecture. The IETF SRv6 Network Programming draft [8] extends the SRv6 specifications by defining new instructions and elaborates the concept of SRv6 SID.

SRv6 segments are regular IPv6 addresses, and since these addresses are not a scarce resource, [8] suggests that SRv6 SIDs can be encoded as `LOC:FUNCT`. `LOC`, for *locator*, corresponds to the L most significant bits of an IPv6 address, and `FUNCT`, for *function*, is the $128-L$ least significant bits. The locator represents a single SRv6 node. The node should advertise itself with the prefix `LOC::/T`.[6] The function is an identifier mapped to a network function installed in the node.

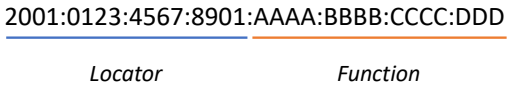


Figure 2.2: Example of a SRv6 SID encoding.

Several SRv6 functions are defined in [8]. These functions extend the regular SRH processing of endpoint nodes described in listing 1, which we name the `End` function from now on. For the sake of brevity, only the functions implemented in our programmable interface (see chapter 4.1) are hereby introduced.

6. In SRv6, since SIDs are regular IPv6 addresses belonging to a node, the node is supposed to advertise its SIDs into an Interior Gateway Protocol.

The following functions all take a single parameter. Each installed SRv6 SID must be mapped to the precise instruction to execute, along with the parameter associated to the instruction. The complete specifications of these functions are available in appendix B. Illustrations of these functions are provided in figure 2.3.

- `End.X`: *Endpoint with Layer-3 cross-connect.* This the SRv6 instantiation of an Adj-SID. After having updated the DA to the next segment, forward the packet to the IPv6 nexthop bound to the SID.

- `End.T`: *Endpoint with specific IPv6 table lookup.* Identical to `End`, except that the lookup for the next segment is performed in the IPv6 routing table bound to the SID, and not the default one.

- `End.B6`: *Endpoint bound to an SRv6 policy.* Insert a new SRH on top of the existing one. The segment list of the inserted SRH is bound to the SID. Forward accordingly to the first segment of the inserted SRH.

- `End.B6.Encaps`: *Endpoint bound to an SRv6 encapsulation policy.* Encapsulate the IPv6 packet in an outer IPv6 header containing a SRH. The segment list of the encapsulated SRH is bound to the SID. Forward accordingly to the first segment of the encapsulated SRH.

- `End.DT6`: *Endpoint with decapsulation and specific IPv6 table lookup.* If a SRv6 packet reaches this SID with `SL = 0` and contains an inner IPv6 packet, decapsulates the outer IPv6 header. The inner IPv6 packet is then forwarded using the IPv6 routing table bound to the SID.

In addition to these endpoint functions, two other transit behaviors are introduced. Transit behaviors are actions performed by transit nodes on IPv6 packets. The routing table of these nodes can be configured to modify the transit behavior for given prefixes or destinations, instead of executing the regular IPv6 forwarding. These actions are executed regardless if the packet contains a SRH or not. Both behaviors are illustrated in figure 2.4.

- `T.Insert`: *Transit with insertion of an SRv6 Policy.* Insert a SRH in the IPv6 packet. Set the IPv6 DA to the first segment. Forward along the shortest path to the DA.

- `T.Encaps`: *Transit with encapsulation in an SRv6 Policy.* Encapsulate an outer IPv6 header with a SRH. Set the outer IPv6 DA to the first segment of the SRH. Forward along the shortest path to the DA.

`T.Insert` and `T.Encaps` are the equivalent of, respectively, `End.B6.Encaps` and `End.B6.Encaps`, without the SRv6 endpoint mechanism.

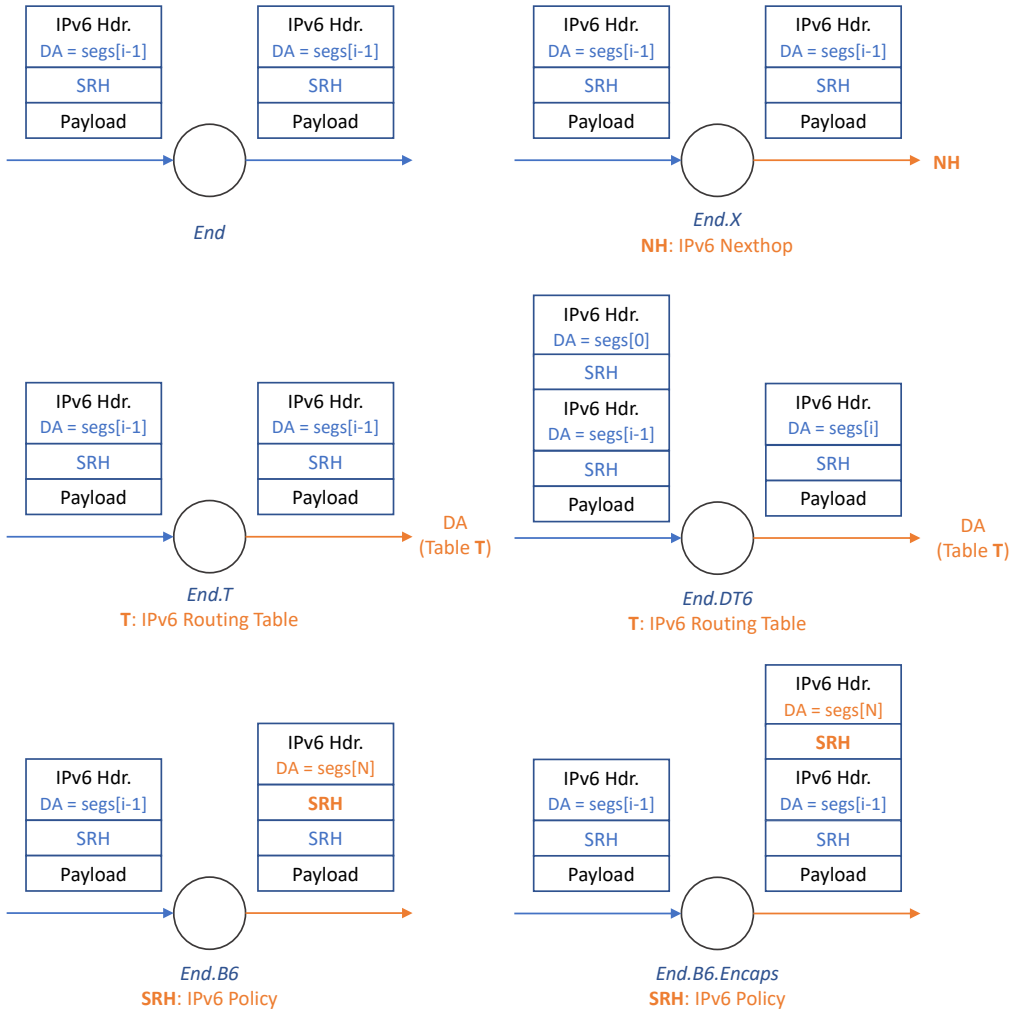


Figure 2.3: Illustration of the principal SRv6 network functions.

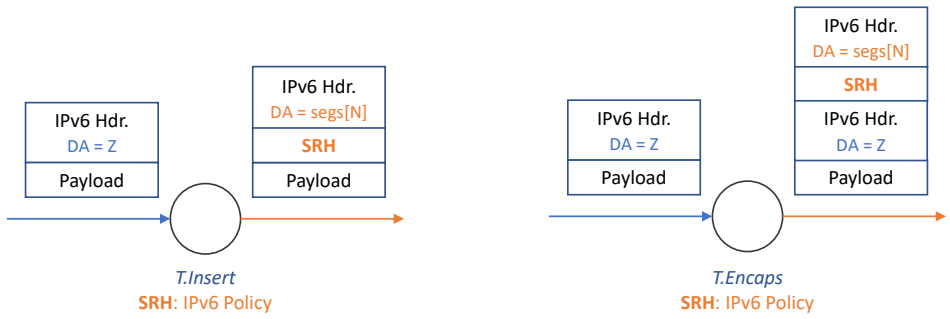


Figure 2.4: Illustration of the `T.Insert` and `T.Encaps` transit behaviors.

2.3 SRv6 implementation in the Linux kernel

The SRv6 mechanisms previously introduced have been implemented in the Linux kernel. Initial SRv6 support has been available as of Linux 4.10, released in February 2017. This section reviews how the specifications of the IETF SRv6 drafts [9] and [8] have been implemented in the kernel. Some key concepts of the Linux network stack are first introduced. We only summarize the key

principles of this implementation, a more complete description is available in [12] and [13].

2.3.1 Packet processing in the IPv6 layer

The network stack is a complex subsystem of the Linux kernel. It implements dozens of networking protocols. In the kernel, all packets are represented as socket buffers, or `skb`'s. A `skb` is an instance of the structure `struct sk_buff`, and embeds the content of a single packet, with its payload as well as its metadata (e.g. checksum, ingress interface). This structure is used throughout all implementations of protocols in the network stack. This stack is divided into multiple layers, ranging from network drivers, that are close to the hardware, to user interfaces, such as the socket API. We hereby describe the implementation of the IPv6 layer and how SRH processing has been added in this layer.

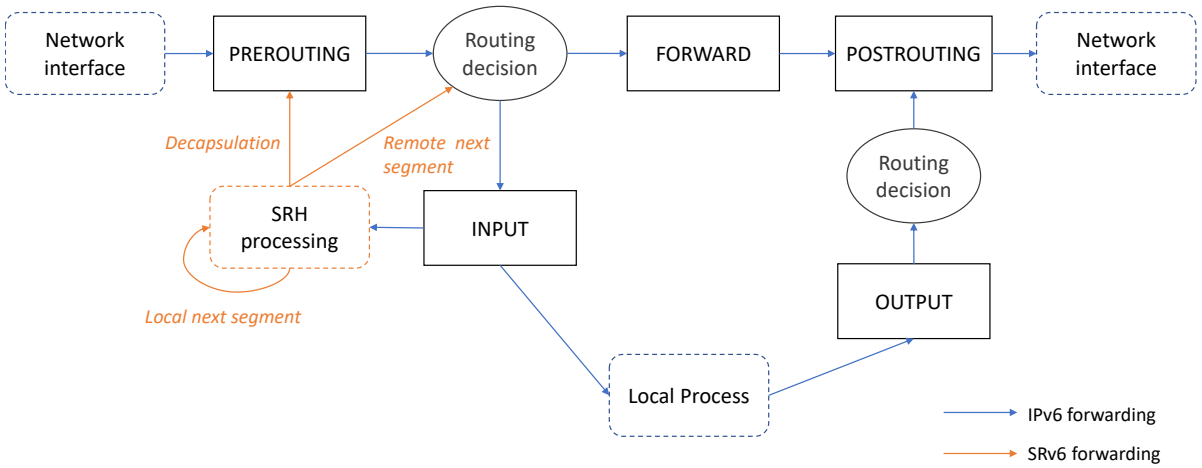


Figure 2.5: Linux IPv6 routing decision process, along with the mechanisms required for SRv6 forwarding.

The routing decision process in Linux is illustrated in figure 2.5. It is a generic process that has been applied to the IPv4 and IPv6 layers. Every IPv6 packet originated from the network driver arrives in the `PREROUTING` stage. In this stage, a routing decision is taken via a longest prefix match of the IPv6 DA in the routing table installed by the control plane. A decision is made, whether the packet is bound to the host or should be forwarded to an egress interface. In the former case, the `skb` enters the `INPUT` stage and is delivered to the next layer (e.g. TCP, UDP, ...). Otherwise, the packet continues to the `FORWARD` level.

Packets originating from higher layers, e.g. because an application sends packets using an UDP socket, directly arrive to the `OUTPUT` stage. Subsequently, a FIB lookup against the destination is done to determine the corresponding egress interface. Packets from the `FORWARD` and `OUTPUT` levels then both reach `POSTROUTING`, where the `skb` is finally sent to the driver of the egress interfaces obtained via the FIB lookups.

This process has been extended in Linux 4.10 to add the support for SRH processing. When an IPv6 packet with SRH reaches an endpoint node, the DA is set to a local address belonging to the node. Following the routing decision process, the `skb` will reach the `INPUT` stage. If `SL > 0`, the packet must be forwarded to the next segment. This is done by decrementing `SL`, and copying

Segments[SL] to the IPv6 DA. This is iteratively repeated until the current segment is a remote segment, and not a local one. The skb is then sent back to the routing process, and a new routing decision is made based on the updated IPv6 header.

If SL = 0 and the header following the SRH is also an IPv6 header, i.e. an inner IPv6 packet is encapsulated, the outer IPv6 header is decapsulated, and the process restarts at the PREROUTING stage, as if the packet came directly from a network interface. If there is no encapsulated IPv6 header, the skb is treated like a regular IPv6 packet and goes to the local processing of the next header, usually of a transport protocol.

However, this extension is only capable of executing the End function. Adding the support for other SRv6 network functions required another infrastructure to be implemented. Moreover, the IPv6 layer by itself does not enable to possibility to inject SRHs in existing traffic. The seg6 lightweight tunnel has been designed to this end.

2.3.2 Insertion and encapsulation of SRHs

The T.Insert and T.Encaps transit behaviors have been implemented in the kernel using a *lightweight tunnel* (LWT). LWTs are a generic infrastructure in the networking subsystem that allows to implement interfaceless network tunnels (the first use of LWTs was to implement MPLS-over-IP tunnels).

All implementations of protocols in the network stack have an input and output function, which are called respectively in the INPUT and OUTPUT stages of the generic routing process. Each route installed in the kernel, whatever its protocol, contains two pointers towards an input and an output function. Usually, these pointers are associated to the functions of the corresponding network layer, e.g. ip6_input, ip6_output or ip6_forward for IPv6 routes.[7] The input and output functions of a route are used to continue the routing process of a skb after the FIB lookup. LWTs leverage the idea that these functions can be replaced by custom ones. This allows to tweak the INPUT, FORWARD and OUTPUT stages of any protocol.

The seg6 LWT has been designed with the goal to implement SRv6 transit behaviors [12]. When creating an IPv6 route using rtnetlink[8], one can specify to use the seg6 LWT. This LWT replaces the input and output functions by seg6_do_srh, which adds a SRH to the packets. Once the SRH is appended to the skb, the original ip6_forward and ip6_output functions are subsequently called by seg6_do_srh.

Additional state needs to be specified when installing a seg6 route, i.e. the type of transit behavior to execute for this route, and the parameters required by the transit function (e.g. the segment list). T.Insert and T.Encaps are available in seg6. Two examples of their setup using iproute2 are provided in listings 2 and 3.

Finally, a user interface using the setsockopt syscall is available, using the option IPV6_RTHDR.

7. Local IPv6 routes that deliver packets to further local processing have an input pointer towards ip6_input. Non-local IPv6 routes that redirect packets to an egress interface have an input pointer towards ip6_forward.

8. rtnetlink allows to read and modify routing tables. It is based on the netlink protocol which enables communication between kernel and user spaces. Several implementations of rtnetlink exist, iproute2 is the standard CLI tool to configure routing tables and other network parameters in modern Linux distributions.

Listing 2 Setup of an IPv6 route with the `T.Insert` behavior using `iproute2`.

```
ip -6 route add dead:beef::/48 encap seg6 mode inline segs fc00::1,fc00::2,fc00::3
    dev eth0
```

Listing 3 Setup of an IPv6 route with the `T.Encaps` behavior using `iproute2`.

```
ip -6 route add fd00:1234::/64 encap seg6 mode encap segs 2001::1,2001::2 dev eth2
```

This allows a user space application to inject SRHs in all the packets originating from a socket.

2.3.3 SRv6 network programming via `seg6local`

A second LWT infrastructure, named `seg6local` [13] has been implemented to support the SRv6 network programming functions defined in [8]. This infrastructure has been released with Linux 4.14.

`seg6local` allows to install SRv6 SIDs mapped to specific functions. Whereas the SRH processing inside the IPv6 layer described in 2.3.1 is only capable of executing the `End` function, `seg6local` allows to use, as of Linux 4.17, nine different SRv6 network functions. `seg6local` behaves similarly to `seg6`. The creation of a `seg6local` route requires two additional arguments: the action[9] to execute, and the parameter required by the action. An example is provided in listing with the `End.X` instruction.

Listing 4 Setup of a SRv6 `End.X` SID using `iproute2`.

```
ip -6 route add dead:beef::/48 encap seg6local action End.X nh fc00::42 dev eth0
```

When installing a `seg6local` route, the LWT modifies the `input` function, such as it points to the `seg6_local_input` function. `seg6_local_input` is responsible for calling the action mapped to the SID, along with its specified parameter.

2.4 Work related to SRv6 network functions

This thesis focuses on implementing an interface for programmable SRv6 network functions in the Linux kernel. SRv6 is an emerging technology, and research on this topic is still in its infancy. We hereby provide a survey of the scientific literature on SRv6 network functions.

Another open-source implementation of SRv6 is available in the fd.io VPP project [14]. Like the `seg6local` LWT, it supports nine SRv6 network functions, but provides a *LocalSID development framework* which can be used to implement new SRv6 network functions. [15] suggests a new

9. In the `seg6local` infrastructure, network functions are named *actions*. Both terms will be used in this thesis, but they refer to the same concept.

transit behavior to design network load-balancers for the application layer. A HTTP load-balancer is implemented as a VPP plugin, steering requests through a chain of candidate servers using SRv6.

[7] and [16] propose two different architectures for SFC SRv6 functions on top of the Linux SRv6 implementation. The former rely on a custom kernel module acting as a netfilter hook to intercept SRv6 traffic and redirect it to virtual containers implementing the specific network function. The latter leverages the `seg6local` LWT and adds a new network function `End.VNF` to direct the traffic to a user space TCP proxy performing the specific network function.

3 BPF, a virtual machine inside the kernel

BPF, for *Berkeley Packet Filter*, originated in 1992 from the need for an efficient packet filter architecture [17]. The Unix versions of the time provided networking monitoring tools with packet capture facilities. Yet, these tools ran in user space. This required all packets to be copied from kernel space to user space first, and only then could the filtering take place. BPF had been suggested as a pseudo-machine language to allow the filtering to directly take place in the kernel, avoiding useless packet copies.

The original BPF architecture comprised a simple instruction set, a compiler translating high-level filtering policies to bytecode and an interpreter running directly in the kernel. It was first designed for BSD, and then ported to Linux a few years later. Most implementations of BPF provided also a Just-in-Time (JIT) compiler, translating the BPF bytecode to native machine code at the execution, which yields better performance than an interpreter.

In 2014, the BPF implementation in Linux has been completely revamped and considerably improved, leading to the inception of *extended BPF* (eBPF). [18] Although the original cBPF (classic BPF) was initially designed as a packet filter facility only, eBPF is a more powerful and general-purpose virtual machine. Its objective is to provide a programmable interface to adapt at runtime kernel components to user-specific behaviors. Several different kernel subsystems adopted eBPF interfaces throughout the last years. It is primarily used in components of the network stack, such as Linux's traffic classifier (`tc`) [19] or the eXpress DataPath (XDP) [20], but its usage is emerging in other parts of the kernel as well, e.g. the tracing subsystem.

eBPF has become the universal virtual machine inside the Linux kernel, and it is expected that more kernel subsystems will start relying on eBPF in the future. This chapter gives an overview of the BPF[1] ecosystem. Its architecture is first described, along with the development process of BPF programs. The mechanisms allowing the virtual machine to interact with the kernel and user spaces applications are then explained. Finally, we introduce the BPF lightweight tunnel infrastructure and the BPF compiler collection.

3.1 The eBPF infrastructure

Designed in 1992, the infrastructure of cBPF, implementing a 32-bit RISC instruction set architecture (ISA), did not keep up with the evolution of CPU design. Modern processors now embed several cores, rely on 64-bit registers, and use complex ISAs with many more instructions, such as atomic operations. Since the original BPF pseudo-machine could not leverage these new features, the eBPF infrastructure has been introduced in 2014 with Linux 3.15.

1. Starting from now on, we use the term "BPF" to refer specifically to eBPF.

This new architecture extends cBPF by moving to 64-bit registers, increasing the number of available registers from 2 to 10 and adding the possibility to call kernel functions, among others. [21] In general, the eBPF architecture has been designed to be close as possible to native 64-bit ISAs (x64, ARM64, ...). This notably enabled the LLVM project [22] to implement a BPF backend, allowing C programs to be compiled to BPF bytecode.

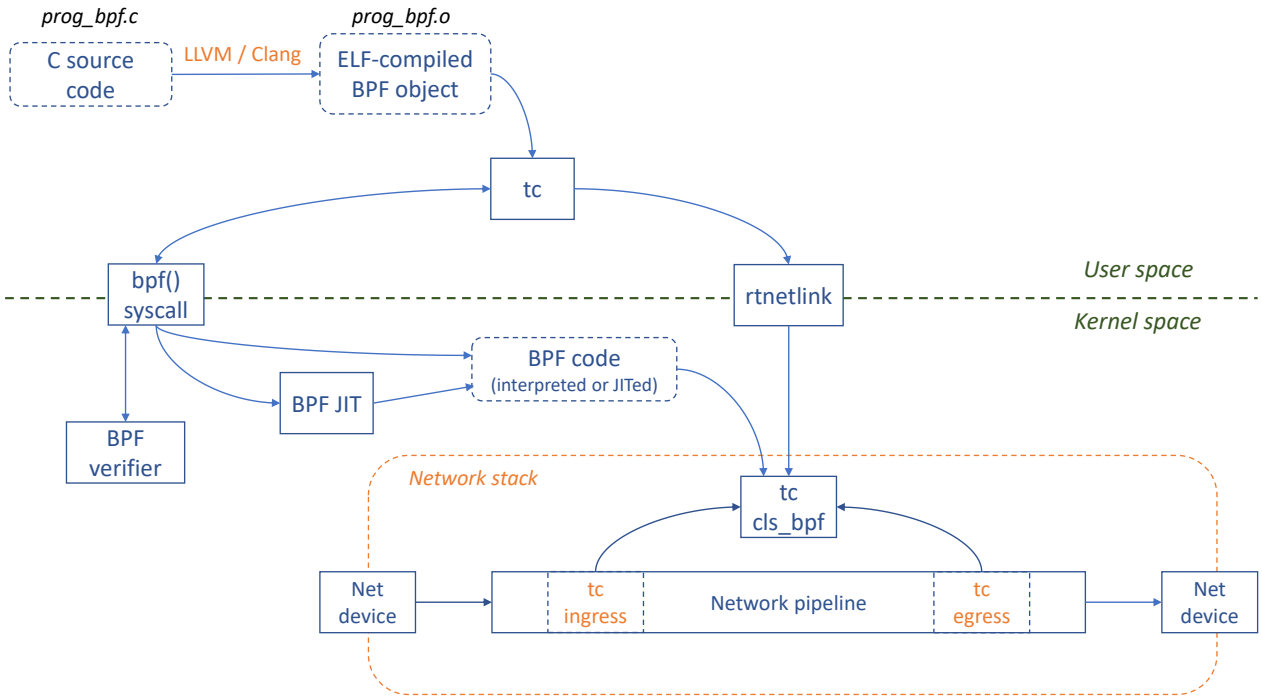


Figure 3.1: Illustration of the BPF infrastructure, used with the `tc cls_bpf` interface.

An illustration of the BPF infrastructure is provided in figure 3.1. BPF programs are attached to a specific code path in the kernel. The choice of the datapath is dependent on the feature being implemented, the example of figure 3.1 focuses on `cls_bpf`, the `tc` interface allowing to develop traffic classifiers in BPF. BPF bytecode is loaded into the kernel using the `bpf()` syscall. Each BPF interface in the kernel defines a program type, identifying in which datapath the program will be inserted, e.g. `BPF_PROG_TYPE_SCHED_CLS` for `cls_bpf`. When loading a BPF program, `bpf()` is called with the program type and the BPF bytecode. The kernel then inspects this code using the BPF verifier.

Running user-specific code poses inherent security and stability risks. The goal of the BPF verifier is to determine if a BPF program could threaten the kernel. A number of checks are performed by the verifier. If a BPF program does not pass them, it is rejected and its loading in the kernel is interrupted. The verifier primarily ensures that the BPF program always terminates, i.e. that no loops could lock up the kernel. A rigorous analysis is also performed on the memory accesses done by the program, i.e. that the stack state is always valid, that no uninitialized registers are read, and that the programs only reads or writes to memory regions it has been given access to. The verifier actually imposes severe restrictions on the BPF code that can be loaded. As a consequence, only a subset of the C language is capable of being compiled to BPF bytecode and being allowed by the kernel. Notably, C constructions such as loops or function calls are prohibited.

BPF programs are loaded by user space applications. In the case of our illustration, the loading is performed by the user space tool `tc`. Once the BPF program is loaded in the kernel, `bpf()` returns a file descriptor associated to the program. This file descriptor must then be passed to the subsystem capable of executing the BPF code. In the case of traffic control, a new classifier `cls_bpf` is created using `rtnetlink`, and the file descriptor is given as parameter of this classifier. For each packet entering the `cls_bpf` classifier, instead of calling a hardcoded function in the kernel, the subsystem executes the associated BPF program. BPF instructions are by default executed using an interpreter.

eBPF JIT compilers are available for several machine architectures. The JIT compilation can be activated by running `sysctl net.core.bpf_jit_enable=1`. This forces an additional step in the loading of BPF programs, as detailed in figure 3.1. Instead of directly storing the BPF bytecode instructions, the kernel executes the compiler and stores the native "JITed" instructions instead. Since the eBPF architecture is very close to the modern 64-bit ISAs, the JIT compilers usually produce efficient native code [18].

3.1.0.1 Example of a `cls_bpf` BPF program

Listing 5 Example of BPF program written in C for the `cls_bpf` classifier.

```
1 __section("my_cls") int cls_main(struct __sk_buff *skb)
2 {
3     uint32_t mark = skb->mark;
4
5     if (mark == 10)
6         return TC_ACT_SHOT; // drop the packet
7
8     return TC_ACT_OK; // classify the packet
9 }
```

An example of program is provided in listing 5. This is a simple BPF traffic classifier dropping all `skb`'s with the mark `10`. Every BPF program written in C is made of a single function, that returns an integer. This function is called for each packet going through the classifier. The function takes as parameter a structure `struct __sk_buff`, which is a proxy structure to the true `struct sk_buff`. Using a proxy structure allows to hide from the BPF program some fields of the `skb`, and to add other fields only relevant for BPF programs. A single BPF object file may contain several functions, the `__section` macro is then used to discern the different programs. If the code of listing 5 is compiled into the object file `my_bpf.o`, the BPF classifier can be inserted into the kernel by running tc filter add dev eth0 parent 1: bpf obj my_bpf.o sec my_cls flowid 1:1.

3.2 Interacting with the kernel using helpers

Although calling user functions is prohibited, BPF code can call *helper functions*, or helpers. Helper functions are functions implemented in the kernel. They act as proxies between kernel space and the

BPF program. Using helpers, BPF programs can retrieve and push data from or to the kernel. The set of helpers that can be called depends on the BPF program type, i.e. the datapath the program is injected into. Some helpers are specific to some program types, whereas others are generic. Two helpers are presented in listing 6. The first helper simply returns a value stored in kernel space. The second one allows to push a VLAN tag into a `skb`. This helper is useful when the BPF interface does not give a write access to the `skb`.

Listing 6 Examples of BPF helpers.

```
// Return the time elapsed since system boot, in nanoseconds.
u64 bpf_ktime_get_ns(void)

/* Push a *vlan_tci* (VLAN tag control information) of protocol *vlan_proto*
   to the packet associated to *skb*, then update the checksum. */
int bpf_skb_vlan_push(struct __sk_buff *skb, __be16 vlan_proto, u16 vlan_tci)
```

3.3 Communicating with user space: maps and perf events

Maps are data structures that reside in kernel space. They allow data to be kept persistent between multiple BPF program invocations, and to be shared with user space applications. From user space, they can be accessed using the `bpf()` syscall.

They are implemented in the kernel as key / value stores [20]. Each map is defined by four values: a type, a maximum number of elements, a value size in bytes, and a key size in bytes. The function helpers `bpf_map_lookup_elem` and `bpf_map_update_elem` can be used to lookup and modify a value associated to a key. Several types of maps are provided, some are generic, others are specific to BPF interfaces and must be used along with their helpers. Generic types include e.g. `BPF_MAP_TYPE_ARRAY`, a simple array implementation, `BPF_MAP_TYPE_HASH`, a hashmap, `BPF_MAP_TYPE_LPM_TRIE` a trie allowing to perform longest prefix match, etc.

Even though helpers are particularly helpful, they are not well suited for event-oriented asynchronous communication between kernel space and user space. Some BPF use-cases may require to send the content of a packet to user space, e.g. for debugging or monitoring purposes. In these situations, it is recommended to use *perf events* [23]. Perf events originate from Linux's performance profiler `perf`. In a networking context, they can be used to pass custom structures from the BPF program to the perf event ring buffer along with the packet being processed. The events collected in the ring buffer can then be retrieved in user space. BPF programs can send perf events using the `bpf_perf_event_output` helper.

3.4 The BPF lightweight tunnel

A lightweight tunnel allowing to execute BPF programs in the routing process of network layers has been added to the kernel in 2016 [24]. This infrastructure is generic and is available for the IPv4,

IPv6 and MPLS layers. When inserting a LWT BPF route, three different BPF programs may be supplied, e.g. using `iproute2`:

```
ip -6 route add fc00::/16 encap bpf in obj bpf_tun.o section hook_in
                                  out obj bpf_tun.o section hook_out
                                  xmit obj bpf_tun.o section hook_xmit
                                  dev eth0
```

In the case of the IPv6 data plane, the BPF programs `in` and `out` are called before, respectively, the `ip6_input` or `ip6_forward`, and `ip6_output` functions. The `xmit` program is called just before yielding the IPv6 packet to the datalink layer (e.g. Ethernet).

Each BPF program of the LWT belongs to another BPF program type, e.g. `BPF_PROG_TYPE_LWT_IN`. The BPF verifier is configured such as each of those program types has different verification rules. Although all programs have the permission to read the `skb`, only the `xmit` BPF program is given a write access (e.g. to insert a L2 header). This restriction is due to stability concerns. When a packet enters the IPv6 layer, its header is parsed and verified. If it is deemed invalid, the packet is dropped. Hence, the subsequent functions `ip6_input`, `ip6_forward` and `ip6_output` all expect a valid IPv6 packet, and do not specify their behavior if called with an invalid packet. Allowing a direct write access to the `skb`'s from the `in` and `out` BPF programs could lead to instability, such as system crashes, if these programs corrupt the packets. Since the BPF verifier is not powerful enough to analyze which BPF programs could potentially corrupt IPv6 packets, write accesses are prohibited for these two program types.

3.5 `bcc`: the BPF Compiler Collection

`bcc` [25], as in *BPF Compiler Collection*, is a toolkit for creating efficient features relying on BPF. `bcc` makes BPF programs easier to write. It provides a C wrapper and directly handles the LLVM compilation process, by taking care of all the miscellaneous details required to compile a BPF program (header files, compilation options, ...).

Moreover, `bcc` includes two front-ends in Python and lua. They allow BPF code to directly be injected into the kernel from these two languages, and provide user friendly interfaces for accessing maps and perf events from user space.

This toolkit is mostly known for performance tracing and monitoring applications, but it can also be used to develop networking features relying on BPF. The daemon architecture that we developed for our SRv6 network functions heavily leverages `bcc` (see 5.3).

4 Unleashing the capabilities of SRv6 through BPF

The network programming capabilities of SRv6 defined in [8] and Linux's corresponding `seg6local` infrastructure are a first step towards giving operators more control on traffic in an IPv6 data plane. Yet, most of the actions of [8] have been designed to meet requirements encountered in already deployed networks, such as MPLS networks connecting different sites of an enterprise through a VPN running on IPv4[1], but do not allow to leverage the new network programming opportunities enabled by SRv6.

It is worth noting that [8] also describes new functionalities relying on SRv6 such as SFC or improved OAM mechanisms, but their definitions in this draft are rather vague. Since such features are very dependent on the underlying technologies, on the devices already deployed in the network and on the actual needs of the operators. Also, no implementations of such new actions have been demonstrated.

Furthermore, at the time of writing (Linux 4.17), all the `seg6local` actions are very rigid. As seen earlier, each `seg6local` route is a relation joining a SID, a specific network programming action and a static parameter associated to this action (e.g. an `End.X` action forwarding all packets going through the SID `2001:db8::2000` to the nexthop `2001:db8::ba1`). This is problematic when one wants to deploy more complex network programming policies, e.g. using the lower 16 bits of the SID as parameter to an `End.T` action. Using the current `End.T` action for this setup requires installing a specific `seg6local` route for each possible parameter value, using many entries in the routing table.

Another mechanism that is cumbersome to implement in the initial `seg6local` infrastructure are actions whose parameters, or even the function to apply, depend on the content of the packet. One could imagine a dynamic `T.Insert` action encapsulating a SRH if the packet matches some criteria, and leaving the packet as-is otherwise, enabling to steer specific traffic flows on different paths (e.g. a VoIP connection would follow a low-latency path, bulk data traffic is forwarded to BGP peers instead of providers, ...). Such setup can already be installed by using multiple `iptables` rules, `seg6local` routes and routing tables, but this method does not scale well in terms of maintenance and performance [26].

Implementing network programming policies which depend in real-time on some state of the system or on the network (e.g. the congestion level of a link could influence the output interface to use in an `End.X` action) is also delicate. A naive solution is to let a daemon replace the `seg6local` routes whenever the state changes and requires a policy update, but this can become very costly when the frequency of updates gets high. A more fine-grain mechanism is needed in these situations.

All of these mechanisms require a more flexible interface to the SRv6 network programming capa-

1. The decapsulation actions (`End.DX2`, `End.DX4`, `End.DT6`, ...) have particularly been designed and advertised in this spirit.

bilities. We only cited here some examples related to actions already implemented in `seg6local`, but many other emerging concepts of network functions leveraging SRv6 cannot be to implemented using the current SRv6 infrastructure inside the Linux kernel: OAM operations using SRH TLVs [27], smart load-balancing of TCP connections to servers [15], advanced Software Defined Network policies [28], Fast ReRoute mechanisms [29], ...

Since it is expected that the interest for more complex SRv6 network functions will grow and because the current `seg6local` framework is not capable of meeting the requirements of such functions, we looked for other solutions up to the task. This problem can be stripped down as a quest for a very flexible programming interface that allows a large range of SRv6 features to be implemented efficiently and in a very accessible way. Ideally, this must be feasible without having to modify the kernel itself, which is a time-consuming, non trivial task that can lead to system instabilities if not carefully executed.

Such technical solution, by its essence, corresponds exactly to what the eBPF virtual machine can provide, i.e. adding at runtime a user-specific feature in some part of the kernel. Custom helpers can then allow BPF programs to interact with any part of the kernel, offering all the flexibility and features needed, while the verifier ensures that the system stability is never compromised. Moreover, the recent developments in the BPF ecosystem, with solutions like `bcc`, enables BPF programs to be quickly developed in C. The BPF JIT compilers also guarantees, for a broad range of architectures, that the BPF code will be executed with native performances. With all these assets, BPF appears to be a perfect solution to our quest.

In the frame of this thesis, we designed and submitted to the Linux kernel maintainers a patchset adding a new BPF interface tailored for IPv6 Segment Routing. This chapter retraces how the specifications of this system were designed, bridging the gap between kernel stability requirements and the need for the most flexible and powerful interface possible. These specifications were then translated into an implementation inside the kernel. Its performances are assessed via micro-benchmarks, followed by a discussion about the functionalities enabled by this interface, and its current limitations. The patchset has been merged in the mainline Linux kernel, and will be available starting from Linux 4.18.

4.1 Design and specifications of a BPF interface for SRv6

Prior to our work, the only BPF interface operating at the network layer was the BPF LWT infrastructure. The `in` and `out` functions allow to process in BPF any IPv6 packet, but they only give a read-only access and no helpers enabling IPv6 functionalities were provided. The `xmit` is called after the routing process and cannot influence the routing decisions. Although this infrastructure offers a framework to possibly implement transit actions in BPF, only firewall and monitoring actions could be implemented due to the lack of helpers.

On the other hand, endpoint functions cannot be implemented using the BPF LWT because they require processing specific to SRv6 (principally ensuring the presence of a valid SRH, and advancing this SRH to the next segment before forwarding the packet). This processing is however already implemented in `seg6local`. With the goal in mind to provide the richest BPF interface possible

for implementing SRv6 network functions, we decided to take the best of both infrastructures and to add in `seg6local` a generic endpoint action executing BPF programs, hence allowing to call user-specific code directly in the IPv6 layer, much like BPF LWT.

Finally, even with the BPF LWT infrastructure and a BPF action in `seg6local`, for implementing, respectively transit and endpoint functions, these interfaces do not provide much functionalities without SRv6 helpers. To provide enough flexibility and functionalities to these interfaces, we also had to design and implement a set of SRv6-specific helpers, allowing BPF network functions e.g. to encapsulate SRHs, modify TLVs or apply basic actions such as `End.X` or `End.T`.

4.1.1 Specifications of `End.BPF`, a BPF interface for `seg6local`

We call our endpoint BPF action `End.BPF`. Like any endpoint function, it can only take place when the current IPv6 DA corresponds to the SID bound to the action. It is also responsible to decrement the `Segments Left` field and to update the IPv6 DA to the next segment in the SRH. Since this processing must be done whatever the goal of the underlying BPF program, it must be performed by the `seg6local` infrastructure itself before executing the BPF code. The BPF program must also have the possibility to indicate that a packet must be dropped instead of being forwarded to the next segment.

When executing the BPF program associated to the action, `End.BPF` must also guarantee, like any other BPF hook, that the stability of the kernel can never be compromised. Since `End.BPF` and LWT BPF share the same architecture, the same security rules that apply to LWT BPF, which have already been described in 3.4, must also be enforced in `End.BPF`. First and foremost, this implies that `End.BPF` can only allow a read-only access to non-sensitive fields of the packet. The BPF program hence needs to resort to specific SRv6 helpers for modifying packets, since only helpers can ensure that the packet integrity is not jeopardised by an invalid writing. If a BPF program corrupts this integrity, the packet must be dropped by `End.BPF`.

Finally, this new action is also responsible for forwarding the packet to the nexthop after the execution of the BPF code. As it will soon be explained, `End.BPF` programs have the possibility to directly influence the forwarding of the packet, e.g. to forward it to a specific next hop or to indicate that the lookup for the next segment must be done in a specific routing table. In these situations, `End.BPF` must respect the indications of the program, otherwise regular forwarding to the next segment is assumed.

4.1.2 BPF helpers for rich SRv6 network functions

Due to the lack of write access to the packet, `End.BPF` only offers a basic interface to the `seg6local` datapath, and it is up to specific helpers tailored for SRv6 to extend the functionalities of our interface. After some thinking about potential use-cases and which features they would require, we considered that the following functionalities should be accessible within the `End.BPF` action: [2]

2. This list only summarizes the possibilities enabled by the helpers in our implementation. More features could be merged in the future.

• Write access to non-sensitive fields of the SRH
• Add, modify or delete TLVs
• Encapsulating a SRH (equivalent of the `End.B6` and `End.B6.Encap` actions)
• Layer-3 cross-connect (equivalent of the `End.X` action)
• Specific IPv6 table lookup (equivalent of the `End.T` action)
• Decapsulating a SRH with `SL=0` (equivalent of the `End.DT6` action)

Table 4.1: Basic features that should be available within `End.BPF`.

Furthermore, since the `End.B6` and `End.B6.Encap` actions are similar to, respectively, `T.Insert` and `T.Encaps` (i.e. they do execute the same operations but not in the same datapath), we will also provide a helper to allow the LWT BPF `in` and `out` hooks to perform these two transit functions. Obviously, most of the classic BPF helpers (access to maps, timestamps, CPU ID, ...) should also be accessible.

The features of table 4.1 have been distributed among several helpers, their specifications have been here summarised below. The full specifications shipped with the Linux kernel are available in the appendix C.

`bpf_lwt_seg6_store_bytes`: write access to non-sensitive fields of the SRH

Despite the fact that BPF programs cannot directly modify packets, some use-cases still require to edit some parts of the SRH. `bpf_lwt_seg6_store_bytes` allows to indirectly store bytes in specific sections of the SRH, which are deemed to be non-critical for the rest of the IPv6 processing if modified to erroneous values. We studied which fields of the SRH should not be editable:

- `Next Header`: must remain constant as its value depends on the structure of the packet.
- `Hdr Ext Len`: must correspond to the actual length of the SRH, its value must remain constant or follow the evolution of the SRH if it has grown or shrunk.
- `Routing Type`: with IPv6 Segment Routing, is always equals to 4.
- `Segments` (and `Last Entry`): for semantic and transparency[3] reasons, the BPF code should not be able to directly edit the list of segments.
- `Segments Left`: the same reasoning as for `Segments` applies here.

Thus, `bpf_lwt_seg6_store_bytes` only allows to modify the remaining fields: `Flags`, `Tag` and the possible TLVs. This is deemed safe, both from Segment Routing's semantic aspects, and from the kernel stability viewpoint (these fields are not critical afterwards in the IPv6 and SRv6 datapaths). On the contrary, write access to these fields could be useful and leveraged by several use-cases (e.g. an OAM SRv6 function indicating the next segment that one of its link is congested by adding a

3. That is, having packets with a constantly changing SRH could be hard to follow for system administrators when troubleshooting their network. Allowing this would also go against the source routing philosophy, where the source defines the hops to go through.

TLV and setting the O and A flags).

This helper is defined very similarly to the well-known `memcpy`, as both do indirect write accesses in memory. `bpf_lwt_seg6_store_bytes` expects an offset corresponding to a byte in the packet (offset 0 indicates the beginning of the packet, i.e. the first byte of the outermost IPv6 header), a pointer to a buffer and its length, and then copies this buffer to the zone in the packet starting by the given offset, while ensuring that this write access is only done on the non-critical parts of the SRH.

`bpf_lwt_seg6_adjust_srh`: grow or shrink a SRH

Even if `bpf_lwt_seg6_store_bytes` allows to modify the content of a TLV, it does not enable the ability to add or delete TLVs by itself. A specific helper is needed to grow or shrink the bottom part of the SRH. It takes two parameters, an offset telling where to grow/shrink the SRH, and the number of bytes that need to be allocated or removed (a negative number of bytes indicates a shrink). This helper must also update the `Hdr Len` field, so that the BPF program does not have to care about this value.

`bpf_lwt_seg6_adjust_srh` is to be used conjointly with `bpf_lwt_seg6_store_bytes` when growing the SRH, since once the room has been allocated for one TLV, it must be filled with its content. Notably, the `Length` field of the TLV must correspond to the size that it actually occupies in memory. It is worth noting that these two helpers could compromise the integrity of the packet if used with incorrect parameters. Whenever they are used, `End.BPF` must afterwards verify that the SRH is still valid. This helper cannot be used to grow or shrink other parts of the SRH, such as the segments list.

`bpf_lwt_seg6_action`: apply generic network programming actions

The remaining features from table 4.1 are all functions from [8] which have already been statically implemented in `seg6local` as actions: `End.X`, `End.T`, `End.B6`, `End.B6.Encap` and `End.DT6`. They all take only one parameter (e.g. the IPv6 nexthop for `End.X`, a SRH for `End.B6`, ...). To avoid having to add a helper for each action separately, we designed instead a unique helper. This helper takes two main arguments, the type of action to apply (e.g. `SEG6_LOCAL_ACTION_END_B6`) and a pointer to a buffer containing the value of the parameter required by the action.

`bpf_lwt_seg6_action` can be called several times inside a single BPF program execution. It is the last valid call to this helper that defines where the packet will be forwarded once the program finishes.

`bpf_lwt_push_encap`: SRH encapsulation for transit functions

Finally, as we previously explained, the BPF LWT `in` hook is the *de facto* mechanism to implement transit functions in BPF. Yet, prior to this work, no specific IPv6 helpers had been designed for

this hook, making it almost useless. We hence introduce `bpf_lwt_push_encap`, which enables to dynamically insert or encapsulate a SRH (with an outer IPv6 header) in a IPv6 packet.

Much like `bpf_lwt_seg6_action`, two arguments must be provided when called: the type of encapsulation to apply (e.g. `BPF_LWT_ENCAP_SEG6_INLINE`), and a pointer to a buffer containing the header to encapsulate (in our case, the SRH). Moreover, `bpf_lwt_push_encap` is designed such as encapsulation for other protocols could be added in the future.

4.2 Implementation and integrity protections

Although the aforementioned specifications are quite brief and not intrinsically complex, we still faced some challenges while implementing `End.BPF` and the helpers in the kernel. Particularly, one of the major issues was to design a mechanism that allows `End.BPF` to detect when the BPF program corrupted the SRH. This section gives an overview of our implementation and also emphasizes on our solution to this integrity obstacle.

4.2.1 Keeping track of the packet integrity between `End.BPF` and the helpers

Since yielding a packet with a corrupted SRH back to the IPv6 layer could be a source of instability in the kernel, `End.BPF` must ensure that the integrity of the packet is maintained after execution of the BPF program. If this is not the case, the packet must be dropped. Following the above specifications, loss of integrity can only arise when the user forges incorrect TLVs. The `Tag` and `Flags` fields can be modified to any value, this does not pose concern with respect to integrity.[4] We hence have to analyze how a BPF program could possibly compromise the validity of the SRH by inserting, editing or deleting TLVs.

First off, the Segment Routing Header is built with the underlying property that `Hdr Len` = $(N_{segments} * 16 + \sum_i^{N_{TLV}} TLV[i].Length)/8$. Since the `Hdr Len` field is automatically updated by `bpf_lwt_seg6_adjust_srh`, the BPF program must ensure that the former relation is always verified by correctly setting the `Length` fields of the TLVs. If not, the TLVs become indistinguishable from one another.

A second source of concern originates from the design of the `Hdr Len` field, which contains the extra length (i.e. the sum of the sizes of the segments and TLVs) of the SRH rounded to 8 bytes. As mentioned earlier, this implies that the total length of all TLVs must be a multiple of 8 bytes. If this property is not respected, a padding TLV must be inserted at the bottom of the SRH. With the specifications of `bpf_lwt_seg6_store_bytes` and `bpf_lwt_seg6_adjust_srh`, when a BPF program wants to add or delete a TLV, it is also responsible for managing the Padding TLV (either by adding one, updating its size, or deleting it). If it fails to do so correctly, the SRH becomes invalid.

Furthermore, one must also observe that when adding or deleting TLVs within the BPF program,

4. Actually, if the user sets the H flag without inserting a valid HMAC TLV, the packet will be dropped by the next receiver, but this does not endanger the kernel itself with the current `seg6local` implementation.

the SRH might temporary be in an invalid state where its length cannot be rounded to 8 bytes without remainder (e.g. starting from a SRH without TLVs, adding a TLV of 20 bytes is invalid until a padding TLV of 4 bytes is appended). In these situations, the true length of the SRH cannot be computed using `Hdr Len` since it is not a multiple of 8 bytes. The `End.BPF` action must hence keep track of the total length of the SRH in bytes separately from `Hdr Len`, as it needs to ensure that the final length is a multiple of 8 bytes, once the BPF code has finished its execution.

This extra state containing the true length of the SRH must be editable within `bpf_lwt_seg6_adjust_srh` and readable from the function calling the BPF program. Yet, in the current BPF implementation, the caller of a program is completely separated from the helpers. The only variable which is shared between both parties is the `skb`, but adding a field to this structure would be an overhead for the entire network stack, even if `End.BPF` is not used on the system. Another naive solution would be to use a global variable to store this state, but several instances of the same BPF program relying on this variable could be executed concurrently in multi-CPUs systems, possibly leading to race conditions. Instead, we use a feature called *per-CPU variables* [30], which instantiates global variables on each CPU, cancelling the risk of race conditions. The structure defined in listing 7 is then stored per-CPU. It can freely be accessed by the program caller and the SRv6 helpers.

Listing 7 Definition of the `seg6_bpf_srh_state` structure.

```
struct seg6_bpf_srh_state {
        bool verified;
        u16 hdrlen; //value of the SRH extra length in bytes
};
```

Moreover, there is also no point in re-validating a SRH which has not been modified by the helpers. This implies that another bit of state must be allocated to know if the BPF program used a helper that could violate the integrity of the SRH. Any call to `bpf_lwt_seg6_adjust_srh` or `bpf_lwt_seg6_store_bytes` sets `verified` to 0, and the former also updates the value of `hdrlen` when it has grown or shrink the SRH.

Using this state, the `End.BPF` implementation can re-validate the SRH after the execution of the BPF program if needed (i.e. if `verified` is set to 0). It first ensures that `hdrlen` is a multiple of 8, and then uses the kernel function `seg6_validate_srh` to determine if the sum of the TLVs `Length` fields equals `hdrlen`, minus the space taken by the segments.

Finally, this validation is also done before executing any action through `bpf_lwt_seg6_action`, as some actions require a stable SRH (e.g. `End.B6`, otherwise a correct SRH could be stacked on an invalid one). The validation will then not be performed again by `End.BPF` if the SRH is not altered anymore after calling the action.

4.2.2 Overview of the implementation

Having discussed the required mechanisms needed to track the integrity of SRHs, we now introduce a global overview of the `End.BPF` implementation, and how these mechanisms have been conciliated

with the specifications of `End.BPF`.

The `End.BPF` action is implemented in a new function `input_action_end_bpf`, located in the same source file as the other `seg6local` actions. Similarly to them, an `End.BPF` action can be set through the `rtnetlink protocol`, e.g. via `iproute2`: [5]

```
ip -6 route add fd00::2 encap seg6local action End.BPF obj oam_ecmp_bpf.o sec
    OAM_ECMP dev eth1
```

Besides the integrity mechanisms, per the specifications, `input_action_end_bpf` must also behave like a regular endpoint function (especially advancing the SRH and forwarding the packet to the next segment notably, cf. listing 1), and dropping the packet if the BPF program requested it. Another situation worthy of note is when the BPF program called `bpf_lwt_seg6_action`, e.g. with `SEG6_LOCAL_ACTION_END_X`. In this case, the special FIB lookup from the `End.X` action has already been executed by the helper and its result is stored in the `skb`. Whenever this happens, it is important that `End.BPF` does not execute the default lookup to the next segment afterwards, otherwise the destination set by `bpf_lwt_seg6_action` would be overwritten.

To properly manage these situations, the BPF program hence needs to communicate with its caller. This can easily be done using the return value of the program, which in our implementation may return 3 values, indicating what should be done with the packet:

- `BPF_OK`: a regular FIB lookup must be performed on the next segment.
- `BPF_REDIRECT`: indicates that the default endpoint lookup must not be performed, and that packet must be forwarded to the destination already set in the `skb`. It is used when `bpf_lwt_seg6_action` has been called.
- `BPF_DROP`: the packet must be dropped.

Starting from the architecture of the regular endpoint function, adding the mechanisms needed to analyze the integrity of the SRH and the handling of the different return values, we came up with an implementation of `End.BPF`, which has been summarized in pseudo-code in listing 8.

Concerning the helpers, besides the integrity analysis routines that have already been described, their implementation is straightforward and is not of great interest. They essentially only act as glue between the BPF code and the kernel functions from the network stack.

Finally, the correctness of this implementation has been asserted with a suite of tests using our homemade SRv6 testing framework segway (introduced in chapter 5) with about forty unit tests. These tests cover all aspects of the implementation, i.e. the well functioning of the helpers and `End.BPF` itself, in both regular and faulty situations (e.g. when the SRH integrity is lost through incoherent calls to helpers).

5. The dev parameter is actually irrelevant and not used here. It is a legacy of `iproute2`.

Listing 8 Pseudo-code version of `input_action_end_bpf`.

```
1  IF NH=SRH and SL > 0
2      decrement SL
3      update the IPv6 DA with SRH[SL]
4
5      create per-CPU integrity state INT_ST:
6          INT_ST[HDRLEN]:= SRH[HDRLEN] << 3
7          INT_ST[VERIFIED]:=1
8
9      run the BPF program, store the return value in RET
10
11     IF INT_ST[HDRLEN] & 7 != 0
12         drop the packet
13     ELSE
14         SRH[HDRLEN] := INT_ST[HDRLEN] >> 3
15
16     IF INT_ST[VERIFIED]=0 and !IS_SRH_VALID(SRH)
17         drop the packet
18
19     IF RET=BPF_DROP
20         drop the packet
21     ELSE IF RET=BPF_REDIRECT
22         forward accordingly to the last FIB lookup performed in the BPF program
23     ELSE IF RET=BPF_OK
24         FIB lookup on the IPv6 DA
25         forward accordingly to the FIB lookup
26  ELSE
27     drop the packet
```

4.3 Performance evaluation

To measure the performance of our implementation of `End.BPF`, we ran several measurement campaigns on Xeon servers. Our setup is composed of 3 servers with Intel Xeon X3440 processors (4 cores and 8 threads clocked at 2.53 GHz), 16GB of RAM and Intel 82599 10 Gbps network interface cards (NIC). The servers are connected in series. The setup is illustrated in figure 4.1. S1 generates SRv6 packets using `trafgen`. The generated packets contain a SRH with two segments, `fc00::2` and `fc00::3`, and a UDP payload of 64 bytes. S3 acts a sink. S2 executes several endpoint functions.

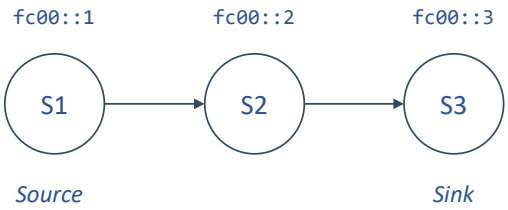
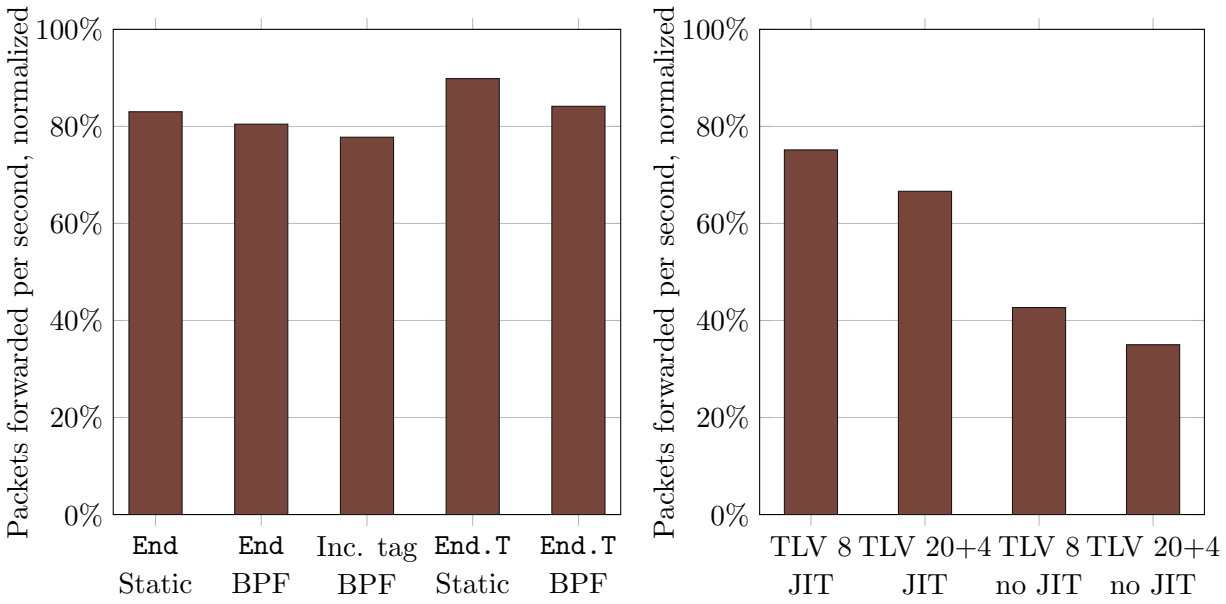


Figure 4.1: Network setup for the evaluation.

In all experiments, unless stated otherwise, the JIT compiler was enabled. For all servers, the interrupts of the NICs were configured such as all traffic was handled by a single CPU core. The results are presented in figure 4.2. All numbers represent the average value of the number of packets forwarded per second, for 20 campaigns where 3 million packets per second were generated. They have been normalized wrt. the performances of pure IPv6 forwarding (610kpps). All throughputs were stable, and had normalized standard deviations below 1%.



(a) Simple endpoint functions

(b) Endpoint functions adding a TLV

Figure 4.2: Performance evaluation of different `End.BPF` functions.

We first developed and evaluated three simple `End.BPF` functions. We implemented BPF equivalents of `End`, i.e. a BPF function doing nothing and of `End.T`, using `bpf_lwt_seg6_action`. A third program BPF fetches the tag in the SRH and then increments it via `bpf_lwt_seg6_store_bytes`. The results of their evaluations are presented in figure 4.2a. The performances of the statically implemented `End` and `End.T` functions in `seg6local` are included as baseline references. Compared

to the static implementation, the BPF equivalent of `End` has a reduced throughput of only 3%. In addition, fetching and incrementing the tag decreases this throughput by 3%. The difference of performances between the static and BPF versions of `End.T` is about 5%. These results lead us to conclude that the `End.BPF` interface has a very low performance overhead.

Moreover, we also developed two more complex BPF programs. The first one adds an 8-byte TLV, and the second one inserts a 20-byte TLV, plus a 4-byte padding TLV. The insertion of TLVs is performed using a specific function of the libseg6 library, introduced in chapter 5. Adding a TLV triggers one call to `bpf_lwt_seg6_adjust_srh` and another to `bpf_lwt_seg6_store_bytes`. Our first program hence calls two helpers, and the second four. The results of their performance evaluation are presented in figure 4.2b. For each function, we did two experiments, one with the BPF JIT compiler enabled, and the other without JIT. The results indicate that adding a small TLV induces only a small throughput decrease compared to a BPF program doing nothing. However, adding a 20-bytes TLV and a 4-bytes padding TLV drops the performance to 75% of the attainable throughput in pure IPv6 forwarding.

Finally, the results of figure 4.2b also allow us to discus the performance impact of the JIT compiler. For both programs, enabling the JIT multiplies the throughput by a factor of 1,8. This factor is expected to increase when the number of instructions per BPF program also increases. A performance profiling on server S2 indicated that when the JIT was disabled, the BPF interpreter occupied about 50% of the ressources of the CPU core dedicated to the traffic handling.

4.4 Conclusion

`End.BPF` is a first step towards a rich and flexible programming interface for SRv6 network functions. Yet the current amount of functionalities offered by `End.BPF` is limited and could considerably be extended. For instance, all actions implemented statically in `seg6local` are not accessible in BPF. Support for them should be added in `bpf_lwt_seg6_action`. Also, even if editing segments is considered a bad policy, it could be interesting to be able to change the segment list of a packet in BPF. A good intermediate and semantically solution would be to pop and push new SRHs in the IPv6 header, the latter is possible with `End.BPF` using the `End.B6` and `End.B6.Encaps` actions in BPF, but not the former. Ultimately, we only considered here features at the IPv6 layer. For instance, SFC functions often require to modify payloads of transport protocols. `bpf_lwt_seg6_store_bytes` could easily be extended to allow the modification of UDP payloads, this is however more difficult to do with TCP connections.

Since the ground work for a rich and flexible programming interface for IPv6 Segment Routing network functions has been laid, we now can leverage `End.BPF` in practice. In the second part of this master thesis, we develop SRv6 networking use-cases which previously required to be implemented in the kernel, in order to determine the strengths and drawbacks of this interface.

Crafting tools for SRv6 BPF

5

SRv6 is still an emerging technology. A direct consequence is that the Linux ecosystem lacks of tools specifically designed for it, although some popular generic networking tools begin to adopt SRv6 support (e.g. Wireshark [31]). When implementing `End.BPF`, we encountered needs that were not answered by existing solutions, and had to developed our own.

Moreover, when designing programs relying on `End.BPF`, we quickly noticed that most of these programs shared some patterns. The components developed in one use-case were often re-usable in others. This led us to craft two libraries that have been used in the development of the programs presented in chapter 6.

These solutions constitute a useful ecosystem for the development of SRv6 BPF network functions. This chapter introduces the motivations that led to their development, and details the key aspects of their implementations.

5.1 segway: a unit testing framework for SRv6

The development of `End.BPF` and of the use-cases relying on it required a unit testing framework to verify the correctness of their implementations. Since no existing solution seemed up to the task, we developed *segway*, a protocol testing framework tailored to our needs, with full SRv6 support.

segway is directly inspired from packetdrill [32], a unit testing framework conceived by Google to make the development and debugging of TCP easier. Designed with the purpose to test network layer protocols, segway allows to send packets to the network stack and to analyze if the kernel forwards them with the expected behavior. Like packetdrill, segway embeds a parser relying on an EBNF grammar, enabling test suits to be written in a convenient syntax. Some examples of unit tests are provided in listing 9.

We implemented segway in Python using several open sources libraries. After having parsed the test suite, segway installs via `pyroute2` [33], a `rtnetlink` client library, a network namespace[1] in the system. It then adds two virtual network interfaces to this namespace: a network layer tunnel `tun0`, and a dummy interface `dum0`. The packets from the unit tests behaving as stimuli are injected through the former, while the latter acts a sink. The injection and the capture of packets is performed using `scapy` [34], a Python library for packet manipulation. A verifier then analyzes if the kernel forwarded the injected packet as expected, and finally indicates which unit tests failed. An illustration of this architecture is available in figure 5.1.

The framework also allows to execute shell commands inside a test suite. By default, only the

1. Network namespaces are logical copies of the network stack. Each namespace has its own routing table, firewall rules, and network devices. Using a network namespace in segway allows to set up a virtual network without modifying the actual routing tables and network parameters of the system.

Listing 9 Examples of simple unit tests for segway.

```
# Routing of a simple IPv6 packet with a UDP payload. The first line represents
# the packet to inject, the second defines the expected one after being forwarded.
> fc00::1 -> fc00::42 / UDP(4242,51) / "foobar"
< fc00::1 -> fc00::42 / UDP(4242,51) / "foobar"

# IPv6 packet with a SRH containing two segments going through an "End" action,
# bound to SID fd00::42. The symbol "+" denotes the current segment.
> fc00::2 -> fd00::42 / [fc00::1,+fd00::42] / TCP(4092, 121) / "lorem ipsum"
< fc00::2 -> fc00::1  / [+fc00::1,fd00::42] / TCP(4092, 121) / "lorem ipsum"

# Encapsulating a packet with an IPv6 header and SRH, no payload.
> fc00::21 -> fd00::9 / [fc00::14,+fd00::9]
< fd00::41 -> fc00::2 / [fc00::1,+fc00::2] / fc00::21 -> fc00::14 / [+fc00::14,fd00
    ::9]
```

`fc00::/16` prefix is routed towards `dum0`, but additional routes, including ones using LWTs such as `seg6local`, can be installed using this feature. It is also possible to specify all the fields of a SRH, including TLVs, and to create new interfaces acting as sinks. Examples of these functionalities are provided in listing 10.

Listing 10 Examples of unit tests leveraging advanced functionalities.

```
# Unit test for an End.BPF action adding an Ingress TLV
`ip -6 route add fd00::5 encap seg6local action End.BPF obj tlv_bpf.o section
    add_ingr_tlv dev dum0`
> fc00::2 -> fd00::5 / [fc00::14,+fd00::5] / UDP
< fc00::2 -> fc00::14 / [+fc00::14,fd00::5] {Ingr: fd00::5} {Pad: 2} / UDP

# Unit test for the End.X action
if add dum1
`ip -6 route add fc42::1 dev dum1`
`ip -6 route add fc01::1234 encap seg6local action End.X nh6 fc42::1 dev dum0`
> dead:beef::2 -> fc01::1234 / [fc00::2,+fc01::1234,dead:beef::1] <tag 20>
< (dum1) dead:beef::2 -> fc00::2 / [+fc00::2,fc01::1234,dead:beef::1] <tag 20>
```

segway has been extensively used in the frame of this work. It not only facilitated the development of `End.BPF` and of the use-cases relying on it, but also considerably accelerated their implementation, since generating and verifying IPv6 packets with SRHs is a cumbersome process without such framework. segway has been released as a standalone open source program under license GPL v3, and is available in the repository of this thesis.

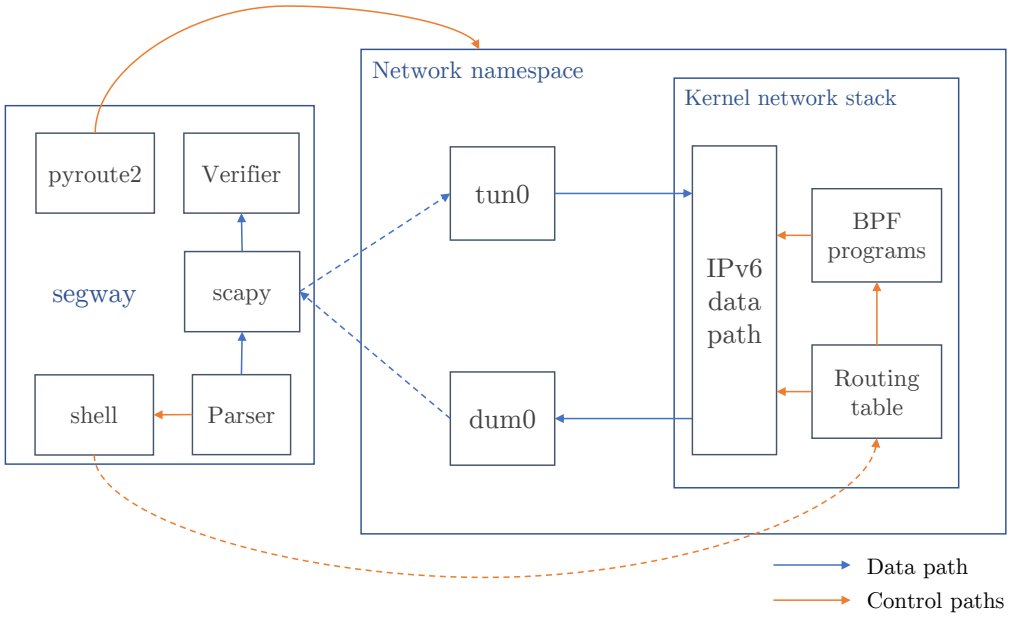


Figure 5.1: Illustration of the architecture of segway and how it interacts with the network kernel components.

5.2 libseg6: a SRv6 BPF library

After having developed several SRv6 BPF programs, it became conspicuous that most of them needed to execute the same type of routines. `End.BPF` actions usually need to retrieve the SRH from the packet. In the case of OAM applications, searching, adding or deleting TLVs is a common requirement. Relying on a library embedding these routines could speed up and facilitate the development of SRv6 BPF functions. With this objective in mind, we crafted *libseg6*, a collection of four useful auxiliary functions, whose specifications are presented in listing 11. Some details are then provided about the motivations for these routines and how they have been implemented in BPF.

`seg6_get_srh`

Although the `End.BPF` action ensures that a valid SRH is present inside the IPv6 header, one cannot assume that the former is directly following the latter. The Hop-by-hop and Destination options might be inserted in between [35]. `seg6_get_srh` inspects the IPv6 header and tries to find the SRH, even if these two options are present.

`seg6_find_tlv`

In an imperative programming language, the TLV lookup operation is implemented using a while loop that inspects each TLV one by one. The termination condition of this loop is to stop when the TLV is found, or when the end of the SRH is reached. However such construction is not authorized by the BPF verifier because the loop formally contains a back-edge. The alternative is to implement

Listing 11 Specifications of the functions of libseg6.

```
/* Return a pointer to the SRH in *skb*, NULL if no SRH has been found. */
struct ip6_srh_t *seg6_get_srh(struct __sk_buff *skb);

/* Search for the first TLV, of the given type *type* and length *len*, in
    the SRH *srh* belonging to *skb*. If a matching TLV is found, return
   its offset starting from *skb->data*, else return -1. */
int seg6_find_tlv(struct __sk_buff *skb, struct ip6_srh_t *srh, unsigned
   char type, unsigned char len);

/* Add a TLV *tlv* of length *tlv_len* at offset *tlv_off* in the SRH *srh
   * belonging to *skb*. Return 0 in case of success. */
int seg6_add_tlv(struct __sk_buff *skb, struct ip6_srh_t *srh, uint32_t
   tlv_off, struct sr6_tlv_t *tlv, uint8_t tlv_len);

/* Delete the TLV at offset *tlv_off* in the SRH *srh* belonging to *skb*.
    Return 0 in case of success. */
int seg6_delete_tlv(struct __sk_buff *skb, struct ip6_srh_t *srh, uint32_t
    tlv_off);
```

a loop with a limited amount of iterations, and to unroll it using the macro `#pragma clang loop unroll(full)`. The number of iterations is a parameter that can be set by the user of the library. Our implementation uses a value of 16, which should be enough in most situations, and do not make the BPF code too large.[2]

`seg6_add_tlv` and `seg6_delete_tlv`

The insertion and deletion of TLVs are performed using the `bpf_lwt_seg6_adjust_srh` and `bpf_lwt_seg6_store_bytes` helpers. These are two similar operations on the SRH that require a proper handling of the padding. As explained in chapter 4, the length of the SRH must always be a multiple of 8 bytes, but no specific rules are enforced on the size of the TLVs themselves. When the sum of the lengths of the TLVs does not round to a multiple of 8, a padding TLV of appropriate size must be inserted. `bpf_lwt_seg6_adjust_srh` only allows to grow or shrink the bottom part of the SRH, but does not deal directly with the padding TLV.

Therefore, it is up to `seg6_add_tlv` and `seg6_delete_tlv`, after having respectively added and deleted a TLV, to correctly adapt the padding. First off, this implies a full iteration over all TLVs to find if a padding TLV is already inserted or not (hence needing another unrolled loop as for `seg6_find_tlv`). Once the target TLV has been added or deleted, a padding TLV may need to be appended, modified or deleted. The handling of these situations require new calls to `bpf_lwt_seg6_adjust_srh` and `bpf_lwt_seg6_store_bytes`, which will induce performances penalties. It is hence recommended to design TLVs whose lengths are multiple of 8 bytes to avoid padding TLVs.

2. The kernel sets a limit on the number of BPF instructions that can be loaded. Using a too large amount of iterations (e.g. 128) made some of our BPF programs larger than this threshold.

5.3 A BPF user space daemon with bcc

Even though the BPF infrastructure allows the execution of user-specific code, it does not provide an environment as rich and as flexible as regular programs enjoy in user space. This quickly becomes an issue when a BPF program needs to interact with components of the system that are not related to the network stack (e.g. writing logs to a file). In these situations, the only solution is to delegate the tasks that cannot be performed in BPF to a user space daemon.

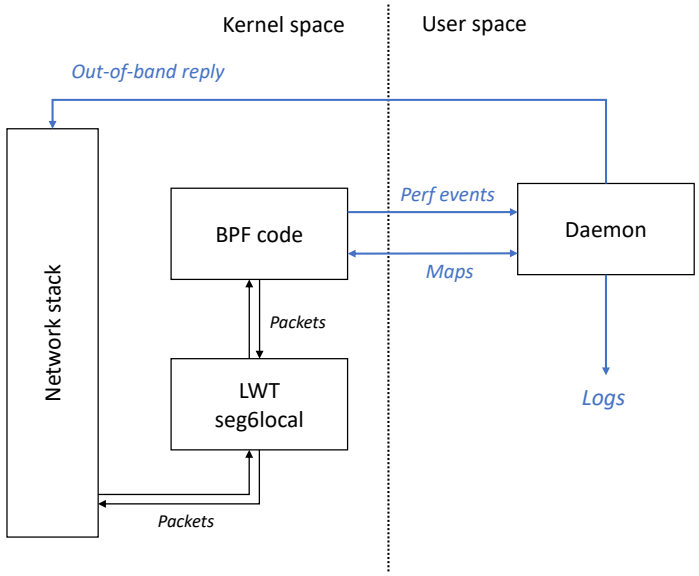


Figure 5.2: Components interacting with a user space daemon tailored for a BPF program. Channels in blue cannot be used without a daemon.

The tasks assignable to a daemon are dependent on the feature being implemented. Figure 5.2 shows two operations that can not be realized purely in BPF, and that have been encountered in the use-cases implemented in this thesis: writing events to a log file and sending out-of-band replies upon the reception of some packets.

To facilitate the development of features requiring user space mechanisms, a daemon infrastructure has been implemented in Python. This infrastructure is responsible for:

1. Via the `bcc` framework, loading a BPF program into the kernel.
2. Using `pyroute2`, inserting an `End.BPF` route with a given SID and linked to the previously loaded BPF code. In this context, `iproute2` cannot be used to create the route.[3]
3. Daemonizing the process.
4. Executing in background the user space tasks required by the use-case.

3. `iproute2` is designed as a CLI tool, and when creating an `End.BPF` route, it takes as parameter the BPF object and then loads it into the kernel itself. This is inconvenient for bcc, as iproute2 does not return the file descriptor of the loaded BPF object, preventing bcc from accessing its maps and perf events, hence the need for pyroute2, which does return the file descriptors.

5. Removing the route when the daemon terminates.

Once the BPF object is loaded in the kernel, bcc allows Python code to access its maps and to listen to perf events. These are the two main channels that can be leveraged to communicate with a BPF program from user space, and are separate mechanisms with different properties. Using perf events, per-packet data can be sent asynchronously from the BPF program to the daemon, and only in this direction.[4] On the other hand, maps can be fetched and modified both in kernel and user spaces. Their usage is more suited for sharing information that is not related to a single packet, but that needs to be kept persistent between several invocations of the BPF code.

User space daemons are key components of the `End.BPF` use-cases that are introduced in chapter 6. The architecture presented in this section has demonstrated itself to be flexible enough to accommodate most of the user space requirements encountered by these use-cases, although some performance issues have been noted in 6.3.

5.4 Conclusion

Using the tools presented in this chapter made the implementation of `End.BPF` and of the use-cases leveraging it much easier and quicker. They constitute a helpful ecosystem for the efficient development of SRv6 BPF features, and we hope that subsequent users of `End.BPF` will also enjoy their usage and adapt them to their own needs.

4. That said, the perf events implementation in the kernel has some security protections to avoid locking CPUs, usually no more than between 25.000 and 100.000 perf events can be sent per second.

6 Leveraging SRv6 and BPF together in practice

The throughput performances of simple `End.BPF` programs have been evaluated in chapter 4. However, these evaluations were restricted to micro-benchmarks and do not indicate how `End.BPF` is capable of performing in more elaborated features.

Raw performance is also not the only interesting metric to evaluate. `End.BPF` being a programming interface for SRv6 network functions, it is important to assess if this interface can actually be leveraged to develop complex functions. The current limitations of `End.BPF` need to be determined, such as future work can be done to extend its capabilities.

To do so, we developed three use-cases. They are solutions to problems which have already been solved by other technologies. They include two different OAM mechanisms, and a data plane feature heavily interacting with user space. The motivations behind each use-case are described, along with their implementations in BPF and of their possible user space mechanisms. Evaluations specific to each use-case are then conducted and their results are subsequently analyzed.

6.1 Passive monitoring of network delays

For network administrators, delay is an important performance metric which needs to be closely monitored. Unusual delays may indicate troubles in the network such as congestion, defective devices or changes in the network topology. Enabling operators to accurately measure delays provides them a better insight into the performance characteristics of their networks and hence facilitates tasks like troubleshooting and performance evaluation [36].

In IP networks, round-trip time (RTT) computation is the most common delay measurement technique, due to it simplicity. The prober only needs to compute a difference between two timestamps, synchronization between local and remote clocks is unnecessary. The most popular RTT measurement tool is `ping`, which measurements the time elapsed between an ICMP Echo Request and the arrival of the corresponding ICMP Echo Reply. However, routing paths are often asymmetric in large networks, inducing forward delays to be different from the backward ones [37]. Hence, one cannot assume that one-way delays (OWD) between endhosts in both directions are each equal to half the RTT. Tools like `ping` thus gives no information about the partition of the delay across the forward and backward delays. To accurately monitor link delays in complex networks, other solutions must be deployed.

Among the recent IETF drafts related to SRv6, [27] specifies mechanisms leveraging the intrinsic properties of Segment Routing to enable efficient and accurate measurements of packet loss, one-way and two-way delays (TWD) between nodes. The delay measurements mechanisms explained

in this draft rely on a *Delay Measurement* (DM) TLV and on the `End.OTP` *"OAM Endpoint with Timestamp and Punt"* suggested in [8], which inserts a timestamp into packets going through SRv6 endpoints. At the time of writing, `End.OTP` is not directly implemented in the Linux kernel, and we implemented a BPF version instead using `End.BPF`.

Moreover, delays measurements are usually performed by actively sending probes into the network, at the price of some traffic overhead. Leveraging the `bpf_lwt_push_encap` helper, we demonstrate that passive monitoring of OWD in a SRv6 network is possible by encapsulating actual traffic in a SRv6 header containing the DM TLV. Experimental results show that such passive monitoring can be performed with almost no performance overhead.

6.1.1 Principles of SRv6 delay measurement

Figure 6.1 shows how delay measurements take place in a SRv6 network, following [27]. An IPv6 packet containing a SRH with DM TLV is sent from the querier (here N1) to an `End.OTP` compliant SRv6 node (N4).[1] The DM TLV contains room for four timestamps: T1, T2, T3, T4.[2] Only the two firsts are used for OWD measurements. The type of measurement (OWD or TWD) is decided by the querier. Upon reception of the probe reply, one-way delay can be calculated following $\Delta T_{\Rightarrow} = T_2 - T_1$[3], and two-way delay using $\Delta T_{\Leftrightarrow} = (T_4 - T_1) - (T_3 - T_2)$.

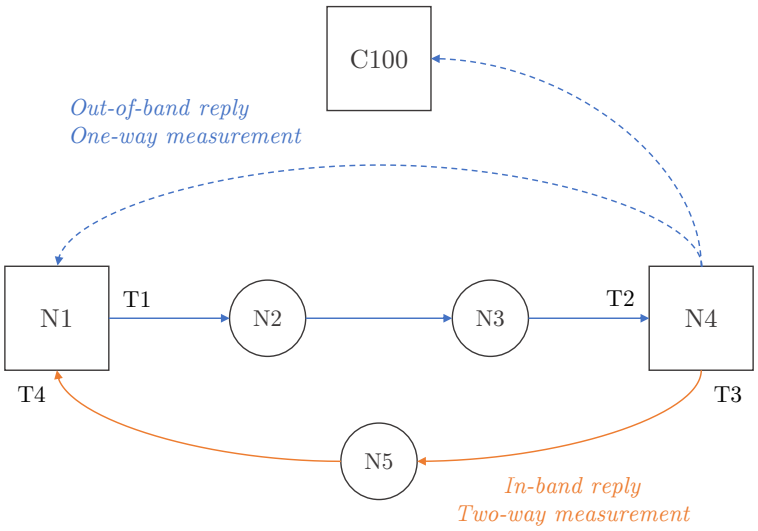


Figure 6.1: Illustration of OWD and TWD measurements. The SRH of the OWD query contains the segments < N2, N3, N4 $_{\text{OTP}}$ >, whereas the SRH of the TWD measurement request forces to follow the segments < N2, N3, N4 $_{\text{OTP}}$, N5, N1 $_{\text{OTP}}$ >. The packet may go through other intermediate routers.

Delays are measured along the path between querier and responder. The followed path can be enforced using segments inside the SRH, as detailed in figure 6.1. It is however important that the

1. This packet may contain no payload and hence only serves to measure delays. In this case we name such packet a synthetic probe.
2. These timestamps represent the time when the packet, respectively, left the querier, arrived to the responder, left the responder and arrived back to the querier. T3 and T4 are not used for a one-way delay measurement.
3. This is valid only if the clocks of the querier and the responder are properly synchronized. The formula for two-way delay does not require this constraint.

segments list does not contain more than one or two `End.OTP` SIDs for, respectively, an OWD and a TWD measure.

The reply to a TWD measurement is always in-band, i.e. the reply is the probe with the T2- and T3-punted DM TLV going back to the querier, whereas OWD measurements replies must be out-of-band. Out-of-band replies are UDP datagrams sent back either to the querier, or to a central controller (C100). The destination to send the reply to is then indicated in a *UDP Return Object* (URO) sub-TLV of the DM TLV (see figure A.4).

The placeholder for the timestamps is the Delay Measurement TLV (see figure A.5). It is injected by the querier and filled by the responder. It contains several fields, and for the sake of brevity, only the most important ones are explained here.[4]

- **Flags**: Message control flags. The most significant bit indicates if the message is a reply (1) or a query (0).
- **Control Code**: 0 for an in-band response requested (TWD), 1 for out-of-band (OWD), 2 if no response requested.
- **QTF, RTF**: Respectively, the timestamp format of the querier and of the responder. Our implementation only supports format 3, i.e. the timestamps consists of a 32-bit seconds field followed by a 32-bit nanoseconds field. As per [36], the responder should always try to use the same format as the querier.
- **Session Identifier**: Set arbitrarily by the querier, identies to which query a reply belongs.

Moreover, to force the timestamping of the DM TLV, the synthetic probe should not be sent to a regular enpoint function, but to an `End.OTP` SID. This function is described in [8] as follows:

Listing 12 Specifications of the `End.OTP` function.

```
1  Timestamp the packet                                    ;; Ref1
2  Punt the packet to CPU for SW processing (slow-path)    ;; Ref2
3
4  Ref1: Timestamping is done ASAP at the ingress pipeline (in
5  hardware). A timestamped packet is locally punted, timestamp value
6  can be carried in local packet header.
7
8  Ref2: Hardware (microcode) only punts the packet. There is no
9  requirement for the hardware to manipulate any TLV in the SRH (or
10 elsewhere). Software (slow path) implements the required OAM
11 mechanisms.
```

It is important to emphasize here that software should not compute the ingress timestamp, since doing so could add to the measurement other significant delays than the ones really belonging to the path between querier and responder (buffering delay due to congestion inside the endpoint, firewall policies, higher priorities tasks locking the CPU, ...).

4. The full description can be retrieved from [27].

Since the definition of `End.OTP` is quite vague regarding the "required OAM mechanisms" to be implemented by software, we define in listing 13 the `End.DM` function as the conjunction of `End.OTP` and the OAM mechanisms needed by the DM TLV.

Listing 13 Specifications of our BPF `End.DM` function.

```
1  Timestamp the packet in HW (TSTAMP_RX_HW)
2  Verify the the presence and validity of a DM TLV inside the SRH
3  Put TSTAMP_RX_HW in DM.T2
4  IF DM.CC == 0x00
5      Put current timestamp in DM.T3
6      Forward to next segment
7  ELSE IF DM.CC == 0x01 and DM.URO exists
8      Send DM to destination contained in URO sub-TLV
9      IF SL == 0
10         IF NH == PROTO_IPV6
11             Decapsulate outer IPv6 header and forward inner IPv6 packet
12         ELSE
13             Drop the packet
14     ELSE
15         Forward to next segment
16 ELSE
17     Drop the packet
```

[27] indicates that T1 and T3 must be computed and added as late as possible in the egress pipelines. This must ideally be done in hardware by the NIC. In our case, as we can only act inside `End.BPF`, the latest opportunity where we can compute T3 and insert it is right before the termination of the BPF program.

6.1.2 Implementation of `End.DM` for OWD and TWD measurements

Our implementation of `End.DM` consists of an `End.BPF` program and a user space daemon based on the architecture described in 5.3. It is solely responsible for timestamping packets sent by a querier (either for OWD or TWD measurements), and does not allow to send new queries or handle replies.

Following our `End.DM` specification, after verifying that the SRH contains a proper Delay Measurement query, we would then have to retrieve the RX (reception) hardware timestamp. These timestamps are stored by the NIC driver in a `skb_shared_info` structure, which is then linked to its corresponding `skb`. This structure is not accessible through direct access within a BPF program, but a BPF helper could easily access it and return the timestamp. Unfortunately, we do not have at our disposal NICs capable of timestamping all incoming packets in hardware, and hence cannot verify the implementation of such helper.

As a fallback solution, we use the RX software timestamps instead. These are computed by the kernel immediately after the CPU fetched the packet via an interrupt or a poll, and stored in the `tstamp` field of `sk_buff` belonging to the packet. We hence added a new helper `bpf_skb_get_tstamp` which returns this field. Yet, relying on a RX software timestamp has two drawbacks that hardware timestamps do not share:

- The RX software timestamp is influenced by CPU load (e.g. interrupts from the NIC missed by the CPU will increase the OWD) and does not represent the precise arrival time of the packet at the router.

- Software RX timestamping is only enabled if at least one socket on the system through requested the software timestamping through `setsockopt`. In our implementation, the daemon maintains an unused socket with `SO_TIMESTAMPING` enabled.[5]

If the DM request asks for an in-band reply, corresponding to a TWD measure, `End.DM` is also responsible for filling the T3 field, and hence needs to have access to the current timestamp. This was not implemented prior to our work, we had to add another helper, `bpf_ktime_get_real_ns`, which is equivalent to a call `clock_gettime(CLOCK_REALTIME, &timestamp)` in user space.[6] The result of this call is then inserted into the T3 slot of the DM TLV, and the packet is forwarded to the next segment.

Finally, for OWD measurements, the reply must be sent out-of-band. The BPF program sends the DM TLV with T2 timestamped to user space through a perf event, along with the mandatory URO sub-TLV. The daemon implemented in Python then fetches each query and sends a regular UDP packet to the destination contained in the URO. This packet contains the DM TLV with T2 filled and the IPv6 and protocol layer headers, including the SRH, enabling network operators to know the source of the probe. If the `End.DM` SID was the last segment in the SRH and contains an encapsulated IPv6 packet, the outer IPv6 header is decapsulated using `bpf_lwt_seg6_action` with `SEG6_LOCAL_ACTION_END_DT6` and forwarded, as detailed in listing 13.

6.1.3 Adding probes on real traffic

Now that we can rely on a sound implementation of `End.OTP`, we can take the OWD mechanism from [27] a step further. Since an OWD probe is entirely contained in the DM TLV, instead of actively sending synthetic probes in the network, we can encapsulate a SRH with a DM TLV on a subset of the traffic and use these packets as probes, as illustrated in figure 6.2. This mechanism has two key advantages: it does not increase the number of packets flowing in the network, and the probing rate is automatically defined by the amount of packets going through the monitored paths.

The key component of this solution is an enhanced `T.Encaps` action inserted at the beginning of the path whose delay is monitored. It encapsulates a defined percentage of the number of incoming packets with a SRH containing a DM TLV and a segment to an `End.DM` SID.[7]

We implemented this enhanced `T.Encaps` action in BPF, using the LWT BPF `out` hook and `bpf_lwt_push_encap`. This BPF program is configured in user space by several maps, providing it the `End.DM` SID, the encapsulation rate and the IPv6 address and UDP destination port needed to fill the URO sub-TLV. It then maintains in a private map a counter tracking the number of packets

5. This is not needed for hardware timestamps. They can be activated through one call to `ioctl` for all packets, and do not require a socket with `SO_TIMESTAMPING` to be created and kept alive.

6. Another helper, available in the mainline kernel, `bpf_ktime_get_ns`, cannot be used it here as it returns a timestamp from the `CLOCK_MONOTONIC` clock, which is system-specific, and is of no use in this situation.

7. Other segments can possibly be added if one wants to combine this solution with another `T.Encaps` SRv6 policy. In figure 6.2, the applied policy is to force traffic going from N1 to N4 to flow through N2.

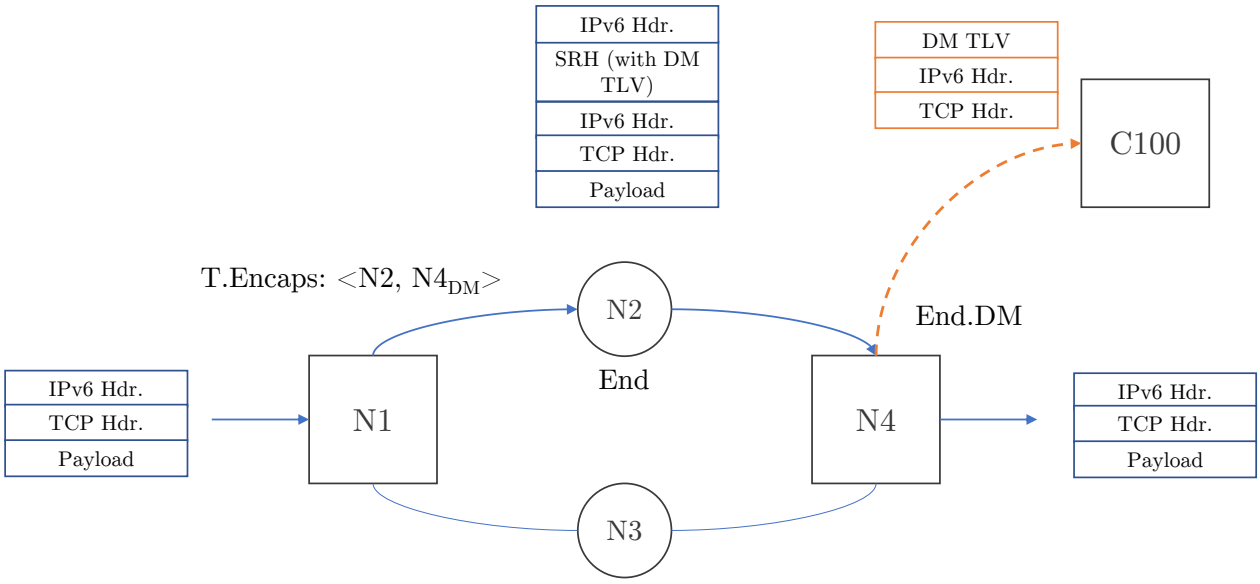


Figure 6.2: Illustration of our mechanism to passively monitor one-way delays on the path N1-N2-N4.

that have been forwarded without encapsulation. This counter is incremented for each packet going through the action.[8] Once it reaches the defined threshold, the current packet being forwarded is encapsulated with a SRH containing the DM and URO TLVs, and the counter resets to zero.

6.1.4 Evaluation of the performance overhead and conclusion

We evaluated the performances impact of `End.DM` and of the enhanced `T.Encaps` action using the setup described in 4.3. S2 executes the `End.DM` and `T.Encaps` BPF programs. S1 uses `pktgen` to generate IPv6 packets without SRH, and `trafgen` for packets with a DM TLV. In all experiments, the JIT compiler was enabled and the interrupts of the NICs were configured such as all traffic was handled by a single CPU core.

The results are presented in figures 6.3 and 6.4. All numbers have been normalized wrt. the performances of pure IPv6 forwarding (610kpps).

These results show that our passive delay monitoring is executed with almost no impact on the forwarding performances, even with a probing ratio of 1:100. We note that our `T.Encaps` action forwards only 5% packets less than the native IPv6 datapath, and 20% packets less than the `seg6` LWT, which does not insert the DM and URO TLVs.

`End.DM` has virtually no impact on performance for a 1:1000 probing ratio and lower, even considering the fact that all packets with a DM TLV are decapsulated. Yet, we noted that our single-threaded user space daemon implemented in Python was not capable of sending more than 7350 probes per second to the controller and consumed 100% of a single core.[9] We did not focus on the performance

8. This counter is always updated using an atomic increment, to avoid race conditions between CPUs. Atomic addition is provided by LLVM through `__sync_fetch_and_add()`, which is mapped to the BPF instruction `BPF_STX | BPF_XADD | BPF_W`.

9. This effect has not been indicated in figure 6.4 because it only appears with a ratio greater than 1:100.

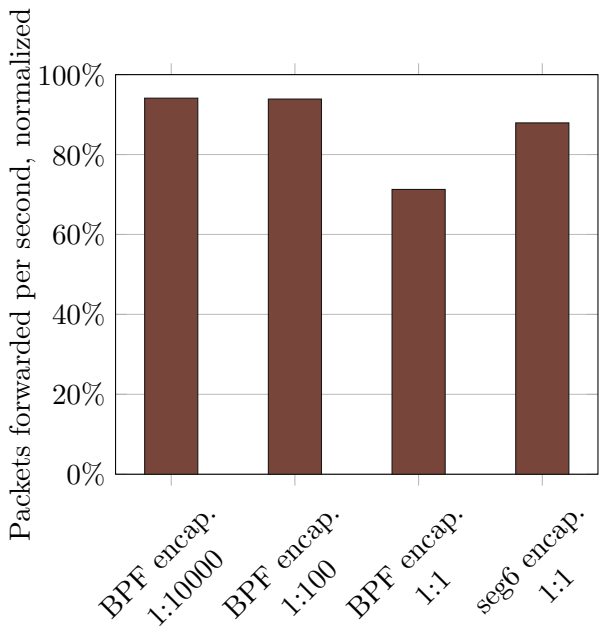


Figure 6.3: Performances comparison of our enhanced `T.Encaps` action for several encapsulation ratios against pure IPv6 forwarding and the encapsulation performed by the `seg6` LWT.

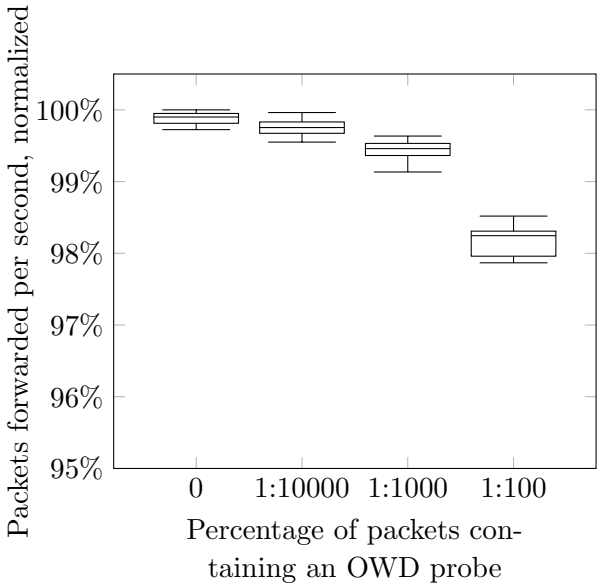


Figure 6.4: Impact of OWD probes going through `End.DM` on the forwarding performances. All OWD probes arrive with `Segments Left=0` and are decapsulated by `End.DM`. The results for each ratio are computed from 20 campaigns of 10 millions of packets.

of the daemon when developing it. An efficient implementation in a lower-level language such as C with a multi-threaded architecture could probably raise this limit much higher.

Finally, from the view point of complexity, the `End.DM` in BPF is implemented using 100 SLOC (Source Lines of Code), including the declaration of the DM and URO TLVs structures (around 30 SLOC). Its user space daemon requires 100 SLOC of Python. Our enhanced `T.Encaps` is written in 130 SLOC (with 60 SLOC for the definitions of the structures and of the maps). These implementations are very compact, hereby illustrating that the BPF ecosystem allows to develop network layer features in a very simple manner.

6.2 Aggregating traffic over multiple links

In 2016, the European Commission pushed a new version of its digital agenda, stating that in 2025, "all European households, rural or urban, should have access to connectivity offering a download speed of at least 100 Mbps [...]" [38]. This 100Mbps bandwidth goal for all customers is a technical challenge for Internet Service Providers (ISP), especially in rural areas, where traditional xDSL cannot scale to such capacities and deploying fiber is too expensive. An emerging solution among ISPs to improve their clients connection is to aggregate several links (e.g. xDSL + xDSL or xDSL + LTE), using technologies such as Multipath TCP (MPTCP) [39] or Huawei's GRE Tunnel Bonding Protocol [40].

Even though these two solutions are used in practice by ISPs, they cannot be deployed on top of

an existing Linux kernel. Huawei's solution is proprietary and closed source. MPTCP is a complex protocol, whose implementation is not trivial, and which must be implemented in kernel space. Notably, the principal Linux implementation has still not been merged into the mainline Linux kernel, essentially because it is tightly linked to crucial and fragile parts of the TCP layer, requiring extensive testing, and also due to its huge code base (~18-20k SLOC) [41].

Since SRv6 inherently allows to control over which path a packet is routed, we wondered if it was possible to leverage SRv6 BPF to design and implement a link aggregation solution with correct performances, capable of running on existing Linux kernels, and with a much more simpler implementation than MPTCP. As answer to this interrogation, we developed a proof-of-concept solution relying on LWT BPF and our implementation of `End.OTP`. Leveraging SRv6, our solution uses a simple per-packet Round-Robin algorithm to distribute the traffic over two links towards a same destination. Its performance has been assessed in two experiments. The results show that the implementation of our proof-of-concept reaches on average 95% of the theoretically attainable aggregated goodput, over links with stable latencies.

6.2.1 Network and architecture design

Figure 6.5 schematizes an example of network topology between an ISP backbone and one of its clients. It is configured to meet the needs of our solution. The Customer-premises Equipment (CPE) is connected to the ISP by two xDSL links. The interface attached to the main xDSL link receives a /64 prefix, whereas the secondary one receives a /112 prefix. Devices inside the LAN are attributed IPv6 addresses from the /64 subnet. We assume that only the /64 prefix is advertised to the Internet by the ISP, the other one does not need to be advertised beyond the aggregator.

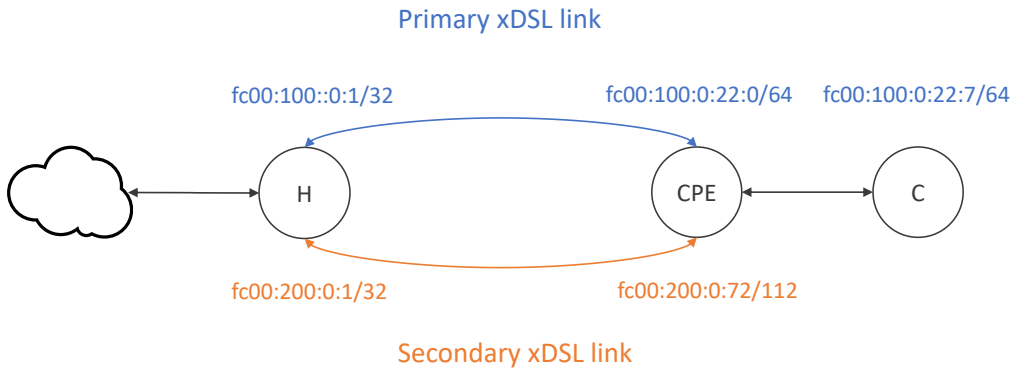


Figure 6.5: Network topology for xDSL links aggregation. All traffic going from or to the client (C) goes through the aggregator (H) inside the ISP backbone, using one of the two xDSL links.

Both the aggregator H and the CPE send packets across the two access networks, the former for downlink traffic, and the latter for uplink traffic. Load balancing is done per-packet instead of per-flow, since per-flow load-balancing, compared to per-packet based policies, leads to both lower network utilization and higher network tail latencies [42]. A major drawback of this approach is that it causes out-of-order packet arrivals at the receivers, since two consecutive packets may travel through links having different delays.

This latter effect is particularly problematic since TCP ensures in-order delivery and uses a buffer

to store out-of-order packets. Upon reception of three out-of-order packets, a retransmission of the packet following the last one that has been acknowledged is triggered by the sender [43]. The sender's congestion window is then reduced, negatively impacting the throughput. Re-ordering can also be detrimental to real-time applications using UDP, for which the reception outdated packets brings no additional useful information (VoIP, streaming, video games, ...). Our solution hence needs to implement a counter-measure to ensure that re-ordering by the endhosts is kept as low as possible, even when the two physical links exhibit different delays.

Furthermore, besides out-of-order arrivals, lack of acknowledgments due to dropped packets also lead to the congestion window to be reduced. These drops usually happen because one part of the forwarding path is congested. Consider a scenario where one wants to aggregate a primary 40Mbps link and a secondary 10Mbps link. With our architecture, the two links are indistinguishable by the endhosts and share the same congestion windows. If the aggregator applies a round robin scheduling policy, the aggregated throughput will never exceed 20Mbps, since sending more than 10Mbps on the secondary link will lead to packet drops, reducing the shared congestion window. This will decrease the aggregated throughput back to 20Mbps, although only a quarter of the capacity of the primary link is used. To reach a global optimum, the CPE and aggregator must hence be able to determine the uplink capacities of each of their interfaces in order to apply an optimal scheduling policy.

6.2.1.1 Load balancing and routing traffic

We use the inherent properties of Segment Routing to balance the client's traffic among the two links. Packets coming from the backbone or from the client's endhost are encapsulated into an IPv6 header with SRH when entering the access network. They are then decapsulated when leaving the access network at the other end. Both the aggregator and the CPE are responsible for encapsulating and decapsulating packets.

All encapsulated SRHs contain two segments. To enforce over which path the packet will be forwarded, the closest router in the ISP backbone connected to the chosen link is inserted as the first segment in the SRH. The second segment is either the aggregator for uplink traffic, or the CPE for downlink traffic. The packets are decapsulated when they reach the second segment. Figure 6.5 illustrates the routing of two packets in the network, over both access links.

Moreover, the aggregator and the CPE must use a scheduling policy to decide through which network each packet must be forwarded. As previously explained, a simple round-robin policy cannot lead to an efficient usage of the capacity of both links if they have different capacities. A reasonable alternative is to apply a Weighted Round Robin (WRR) scheduling algorithm instead, whose weights match the uplink links capacities, as seen by the CPE or the aggregator. The underlying issue of this approach is that it requires to efficiently estimate these capacities.

A possible and probably efficient solution to this issue would be to measure in real-time the maximum attainable throughput on each link, e.g. by analyzing packet drops. In the frame of our proof-of-concept, like [40], we limit ourselves to a simpler approach and propose to collect this information from the physical layer. xDSL modems usually synchronize the link capacities with the digital subscriber line access multiplexer (DSLAM). This information is then often easily obtainable from

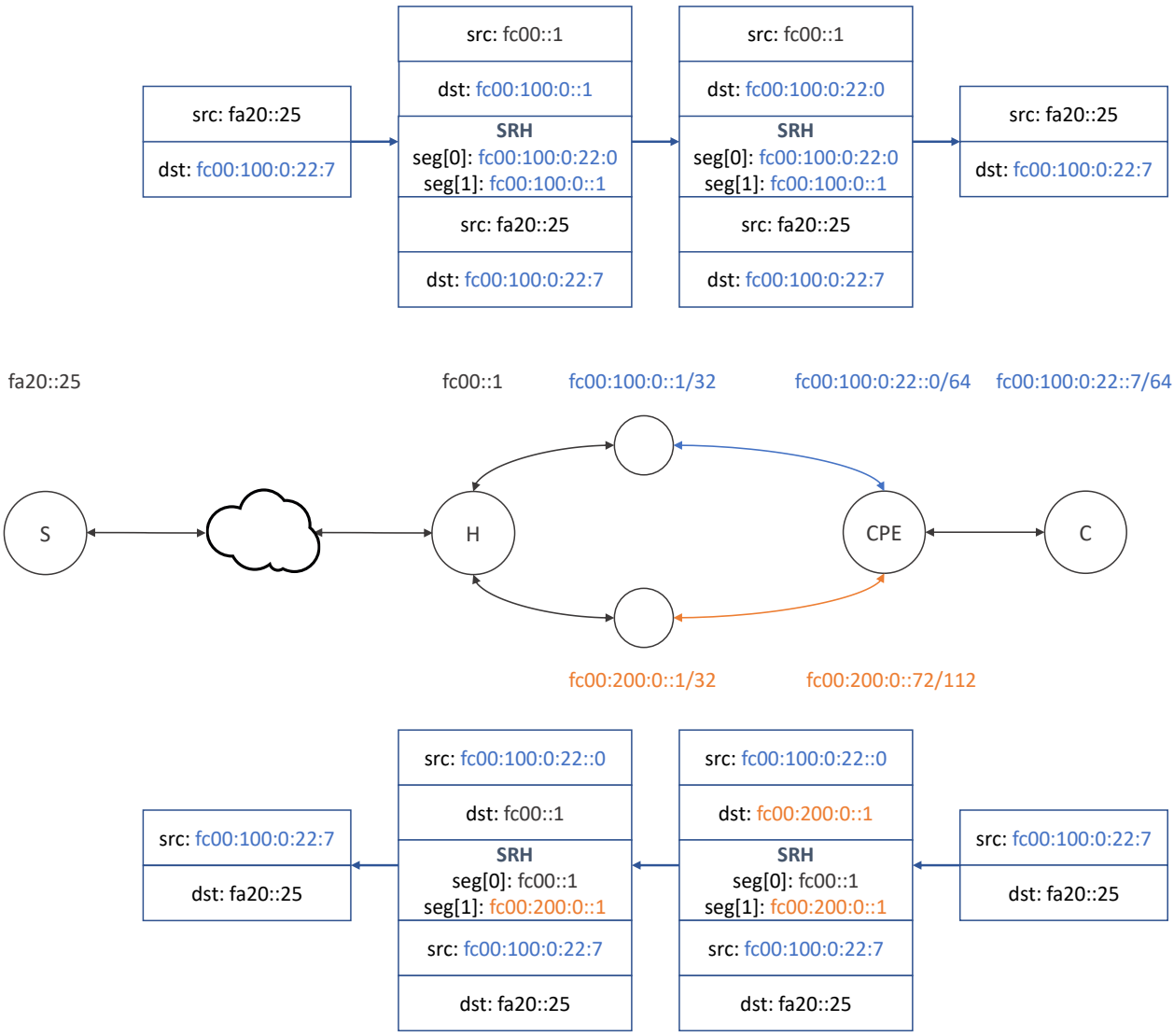


Figure 6.6: Examples of the evolution of summarized IPv6 headers of packets going through the network.

the modem and can directly be used in the scheduling policy. However, this approach cannot detect congestion beyond the xDSL link, e.g. in the backbone, and hence cannot adapt to such situations.

6.2.1.2 Delay compensation at the network layer

Re-ordering is a crucial problem for multipath protocols as both links do not necessarily share the same latency. Multipath TCP solves this problem by resorting to a packet scheduler, which uses a data sequence mapping, associating segments sent on different subflows to a connection-level sequence numbering. This allows packets sent on different paths to be correctly re-ordered upon reception. [35] Segments arrived before their predecessors are stored in a receiver buffer, waiting on the reception of the latter.

One might think about implementing a similar solution in our use-case to mitigate re-ordering by

buffering early-received packets at the aggregator or CPE. Yet this scheme encounters a major BPF caveat: execution time (in terms of number of instructions) of BPF programs are deterministically bounded, and constructions such as infinite loops are prohibited. The only solution in the BPF ecosystem to delay packets by a variable amount of time is to send them to user space using a perf event, delay them in a buffer, and finally send them back to kernel space for transmission, at the price of a very significant overhead.

Another idea is to try to equalize both access links in terms of latency. Instead of using data sequence numbers and ensuring a perfect re-ordering of the packets before sending them back to the destination, the aggregator delays data going through the fastest link by the difference of the links delays. This is a best-effort approach that requires a precise knowledge of the links latencies from the aggregator. It is then be up to the TCP layer on the endhosts to finalize re-ordering if the links delays are not perfectly synchronized, but TCP fast retransmissions should be less likely to occur. If the latencies cannot correctly be estimated (e.g. due to strong jitter), this solution may not be efficient and important re-ordering may still occur.

6.2.2 Implementation

Our implementation is divided into several programs. The main component of our solution are the two WRR load balancers, implemented as IPv6 LWT BPF programs, and installed on both the aggregator and the CPE. The SRH decapsulation is performed by the SRv6 datapath of Linux's IPv6 layer and does not require a BPF action.

Since the aggregator needs to evaluate the delays on the two paths, in both directions, we extended our `End.DM` implementation to be able to respond to Two-Way Delay Measurements queries. A userspace daemon on the aggregator is then responsible for sending and receiving TWD measurement requests, installing the `tc` delay buffers on each direction and updating the compensation delays of these buffers based on the latest delay measures.

BPF Weighted Round Robin load balancer

We implemented the WRR scheduling algorithm [44] in BPF. This implementation is restricted to balance the packets among only two links, allowing the algorithm to be coded without an infinite loop. An extension to N links is also implementable in eBPF and would be trivial to do.

The algorithm requires two state variables to schedule a packet: cw, the current weight in scheduling, and i, the last selected link. Since BPF cannot maintain state between different program invocations, the only way to keep this state persistent is to store it in a map.

A user space program is also responsible for taking the SIDs of the entry-point routers on each link, the weights of those links (i.e. the uplink capacities in Mpbs), and the SID of the proxy at the other end of the access network. The user space program then pushes all these variables through maps to make them available from the eBPF code. The latter then always fetches the whole content of the maps at the beginning of its execution. Moreover, the value of the greatest common divisor (GCD) of the weights is also needed by the WRR algorithm, but computing this value at the beginning

of each BPF program invocation would be computationally intensive, hence it is computed once in user space and also stored in a map.

Upon reception of a packet from the ISP backbone or the client's LAN, the WRR algorithm is executed. After it yielded a scheduling decision, a SRH with two segments is created, the first one enforces the access network the packet will be forwarded. It is then encapsulated using the `bpf_lwt_push_encap` helper, and the packet is released back to the IPv6 layer.

It is worth noting that encapsulating a SRH requires the MTU (Maximum Transmission Unit) between CPE and aggregator to be at least larger than the MTU of the LAN and the ISP backbone, augmented by the size of a SRH with two segments (40 bytes). If this condition is not met, the SRH cannot be inserted as IPv6 does not allow packet fragmentation by an intermediate router. However, such configuration is already standard in networks using aggregators.

Userspace daemon for link delay compensation using `tc netem`

As previously explained, a simple load-balancing solution without latency compensation between links would be inefficient as this could lead to out-of-order arrivals of packets. It is up to our user space daemon to estimate the delays on the links and orchestrate delay buffers for delaying both the uplink and downlink data that travelled along the fastest paths. Adding delay on a link is easy to implement using Linux's `tc` (*traffic control*) tool, whose `netem` module allows to delay transmission of traffic matching given criteria by a specific amount of time. This daemon is a Python program, leveraging the infrastructure detailed in 5.3, and is launched only on the aggregator.

At initialization, the daemon creates for each network interface a `tc` root queuing discipline (qdisc), and for each possible traffic direction (uplink LTE, downlink LTE, uplink xDSL and downlink xDSL), installs a class containing a filter and a `netem` delay qdisc. Each filter ensures that its class only matches traffic going to the corresponding direction, e.g. using a specific mark set by an ip6tables rule.[10]

However, this solution requires to install on the aggregator four triples of a `tc` queuing discipline, class and filter for each connected CPE. It would probably not scale efficiently for thousands of connected CPEs. A new feature in the kernel allowing time based packet transmission is being developed [46]. It could be directly leveraged by our BPF load balancer if an adapted helper is provided, enabling to delay `skb`'s by a given amount of time. This solution would require only one global queuing discipline, instead of currently four per CPE.

In order to measure the one-way delays through the four directions, the daemon sends at frequency f TWD measurement requests through both links to the `End.DM` program running on the CPE.[11] A UDP datagram with no payload is sent from user space. A SRH is inserted to the packet using `setsockopt` and contains a DM TLV. Finally, an `End.BPF` program with a specific SID is also

10. A recent ip6tables feature [45] allows rules to filter on the previous SID the packet went through. In our use-case, one can then easily distinguish incoming packets at the aggregator by looking at the previous SID, which corresponds to the entry-point router. Outgoing packets can be classified directly by looking at the IPv6 destination, corresponding to the first segment.

11. This `End.DM` has its own SID, but this has not been depicted in figure 6.5.

installed by the daemon, this program catches the Delay Measurement replies from the CPE and send the DM TLV to daemon using a perf event.

Instead of directly using the latest values of delay in each direction, the daemon maintains 4 Moving Averages, using the N last delays computed. Based on these averaged delays, the program determines which path, uplink and downlink separately, is the fastest. It then computes the difference of delays, i.e. the compensation delay, between the slowest and fastest link, and sets the `netem` delay policy on the fastest direction to this compensation delay. Averaging delays is a countermeasure to inaccurate measured delays, induced e.g. by jitter, that could lead to massive re-ordering if they were directly used in the computation of the compensation delay.

6.2.3 Performance evaluations and analysis

The performances of this solution have been assessed in two experiments. We configured a network as described in figure 6.7. All links use Ethernet. S1, S2, S3 and S4 are servers with an Intel Xeon X3440 processor ands NICs capable of handling at least 1Gbps. S1 and S4 play the role of endhosts. S2 acts as the aggregator. The CPE is a Turris Omnia router running OpenWRT with a 1.6 GHz dual-core ARMv7 processor and 1Gbps NICs. S3 allows to insert latency on the links and to limit their capacities at a fixed value below 1Gbps, using `tc netem`. S3 connects each link using an Ethernet bridge. The BPF JIT is only enabled on S2, not on the Turris Omnia.

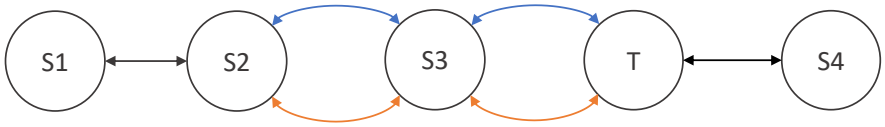


Figure 6.7: Network setup for performance measurements. (T) represents the Turris Omnia. The other nodes are servers.

Evaluation of the forwarding performances

The goal of our first experiment is to study the impact on the forwarding performances of the SRH encapsulation done in BPF and the decapsulation performed by the kernel. UDP flows are generated between the endhosts using `iperf3` with different payload sizes. Smaller payloads induce a larger number of packets sent. The more packets have to be forwarded at the network layer, the more the CPU load of the aggregator and CPE will increase. The underlying objective is to observe to what extent the additional encapsulation and decapsulation routines will affect the CPU load.

S3 is configured to impose no additional delays, nor any capacity restrictions. A baseline reference is evaluated when both aggregator and CPE act as regular routers and only use one Ethernet link. In this configuration, the same performances have been observed in both directions.

The results are shown in figure 6.8. In this experiment, the Turris Omnia is always the bottleneck, either because its interface is limited to 1Gbps or because its CPU cannot handle the packet rate. In downlink, when the CPE is decapsulating and forwarding the packets, the total throughput only decreases by about 10% compared to the baseline performance. For both the downlink and

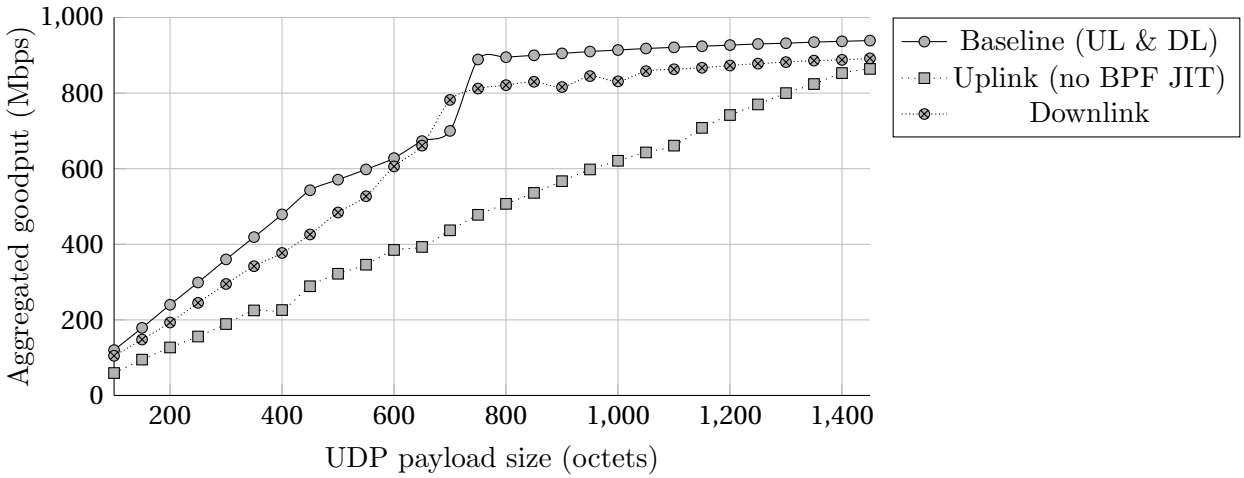


Figure 6.8: Maximum aggregated UDP goodput between endhosts for different payload sizes. Uplink is defined as the direction from the CPE to the aggregator, vice-versa for downlink.

the baseline scenarios, the NIC starts to become the bottleneck when the payload size reaches 750 bytes. Before that, the CPU of the Turris Omnia is the limiting factor.

When transferring data uplink, the CPU is always the bottleneck and is constantly overloaded. That said, our solution still reaches almost the 1Gbps line rate when the payloads are close to 1450 bytes, which corresponds to the usual 1440 bytes value of the Maximum Segment Size in TCP with IPv6. This high load on the CPU is due to the fact that we had to disable the BPF JIT compiler on the Turris Omnia.[12] Yet, it is expected that substantial performances improvements could be obtained in uplink with a functional JIT compiler, possibly reaching similar performances to the downlink scenario. The micro-benchmarks in 4.3 showed that for little BPF programs, the JIT induced a x1,8 speed-up factor, a similar factor can be expected for this use-case. In any case, the current setup is capable of forwarding throughputs of modern access networks (˜100Mbps).

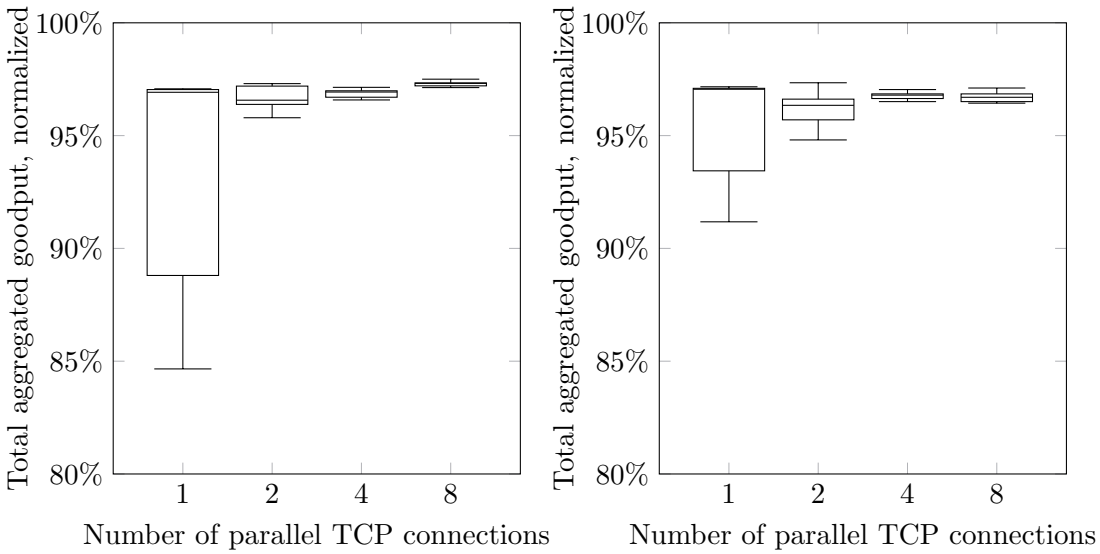
Evaluation of the aggregated goodput

A second experiment has been conducted to validate this proof-of-concept as a whole, by evaluating the aggregated goodput that our solution is capable of delivering.

S3 has been configured to adapt the two links between the aggregator and the CPE, such as they exhibit characteristics similar to actual xDSL links. We call the primary link L1, and the secondary link L2. The delay of both links is configured to follow a normal distribution. L1 has an average RTT of 30ms with a standard deviation of 5ms. L2 has an average RTT of 5ms and a standard deviation of 2ms.

12. Each CPU architecture has its own implementation of the JIT. In the case of the Turris Omnia, running on an ARMv7 32-bit CPU, the JIT compiler is not capable of correctly compiling our BPF program responsible for the WRR scheduling and the SRH encapsulation, although the `End.OTP` BPF program works fine with the JIT. The implementation of the JIT for ARM 32-bits is quite recent (August 2017 [47]) and does not seem to have been extensively tested and used in practice.

This experiment measures the total aggregated goodput observed for an increasing number of parallel TCP connections. The traffic is generated using `nttcp`, S4 acts a sender and S1 as receiver. The moving average of the aggregator has a size of $N = 15$ and the frequency of `End.OTP` updates is $f = 0.5$ Hz. Two scenarios have been considered: symmetric links with 50Mbps uplink capacities, and an asymmetric configuration in which L1 has a 50Mbps throughput, and L2 only 30Mbps.



(a) $C_1 + C_2 = 50 + 50$ Mbps. Normalized by the theoretical limit of 91.15Mbps.

(b) $C_1 + C_2 = 50 + 30$ Mbps. Normalized by the theoretical limit of 73.2Mbps.

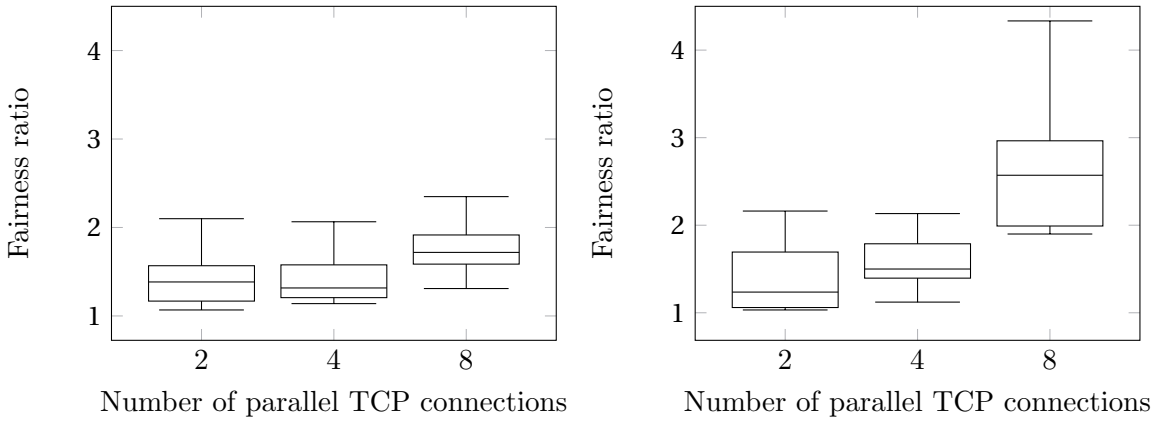
Figure 6.9: Evolution of total aggregated goodput for an increasing number of parallel TCP connections, in both scenarios.

In each scenario, and for each number of parallel connections, 20 campaigns with a duration of 15 seconds each have been launched. The results are available in figure 6.9 and are normalized based on the maximum theoretical goodput attainable in our setup.[13] On average, 96% of the maximum theoretical aggregated goodput is reached, although we observe that the goodput is not always stable when a single connection is fired. In both symmetric and asymmetric scenarios, the total aggregated goodput converges to 97% of the theoretical limit when the number of parallel TCP connections increases, and without stability issues. There seems to be no fight for ressources between connections.

From the same data, we also computed what we define the *fairness ratio* among n parallel connections: $F = \frac{max(G)}{min(G)}$, with G the n-tuple of the connections goodputs of each campaign. Ideally, no TCP connection should be advantaged over another, and F should be close to 1. Figure 6.10 shows that this is not the case. A larger number of parallel connections induces a less fair distribution of the bandwidths. Asymmetric capacities also seem to notably amplify this effect. That said, with

13. The links in black in figure 6.7 have a MTU of 1500, whereas the others have been configured with a value of 1600, to reserve room for the SRv6 encapsulation. In this setup, `nttcp` sends segments of 1428 bytes, the TCP header occupies 32 bytes and the IPv6 header 40 bytes. Since the capacity limit is imposed on the links between the CPE and aggregator, the outer IPv6 header and the SRH must be taken into account as well. The maximum theoretical goodput attainable is hence $\frac{1428}{1560} = 0.915$ the sum of both links throughputs.

$n = 2$ or $n = 4$, F is still close to 1.3 on average, even though we did not design our solution with fairness in mind.



(a) $C_1 + C_2 = 50 + 50$ Mbps. (b) $C_1 + C_2 = 50 + 30$ Mbps.

Figure 6.10: Evolution of the fairness ratio between parallel TCP connections for an increasing number of connections, in both scenarios.

6.2.4 Discussion and possible improvements

We developed using the LWT BPF and `End.BPF` infrastructures a solution for aggregating two access links. Leveraging `bpf_lwt_push_encap`, we implemented a `T.Encaps` action with a Weighted Round-Robin scheduling on top of it. Generally speaking, this use-case demonstrates that dynamic SRv6 encapsulation policies can easily be built in BPF, depending on state pushed from user space if needed. We also re-used our `End.OTP` implementation in a setup where TWD measurements were required.

Our implementation requires only ˜250 SLOC of BPF and ˜400 SLOC of Python. Its performance has been assessed and results show that up to 1Gbps traffic can be distributed over two links using a recent CPE router. For links whose latencies are stable, we showed that this solution could aggregate up to 97% of the maximum theoretical goodput.

Still, our implementation is only a proof-of-concept and the solution is largely improvable. Even if accessing uplink capacities from the physical layer often gives a reasonable estimation of the weights needed by the WRR scheduler, it offers no guarantees that the links are not congested by cross-traffic. In such situations, the scheduling would not be efficient, and possibly leading to a sub-optimal aggregated throughput. Instead of using a WRR scheduler, we should have implemented a *Deficit Weighted Round Robin*, which takes into account the size of the packets. However, the implementation of the DWRR is close to the one of the WRR. The translation of this algorithm to BPF should hence not be an issue. Since we only generated packets of same size in our experiments, the use of a WRR instead of a DWRR is expected to yield the same results.

Moreover, the solution should also measure in real-time if, besides congestion, a link does not exhibit too larges jitters and drop rates. In such cases, the link should be deactivated and all traffic should flow through the remaining one. Taking into account jitter would be a straightforward extension

to our solution, but measuring drop rates requires to implement a new mechanism using TLVs to track the number of packets sent through each link.

Finally, we restrained the evaluation to two links with low jitter values (2ms and 5ms), further work could analyze if our solution is still efficient on links with a high jitter, e.g. WiFi or LTE links.

6.3 SRv6 multipath-aware traceroute

Our `End.OTP` implementation enables a network operator to accurately measure the OWD and TWD on specific paths, but when troubleshooting unusual latencies in complex networks, the delay measure itself is only one aspect of the problem. If the network heavily uses schemes like Equal Cost MultiPath routing [48], traffic between two endhosts can follow distinct paths, and undergo different delays. Identifying which links induce high latencies requires all paths between endhosts to be determined and probed separately.

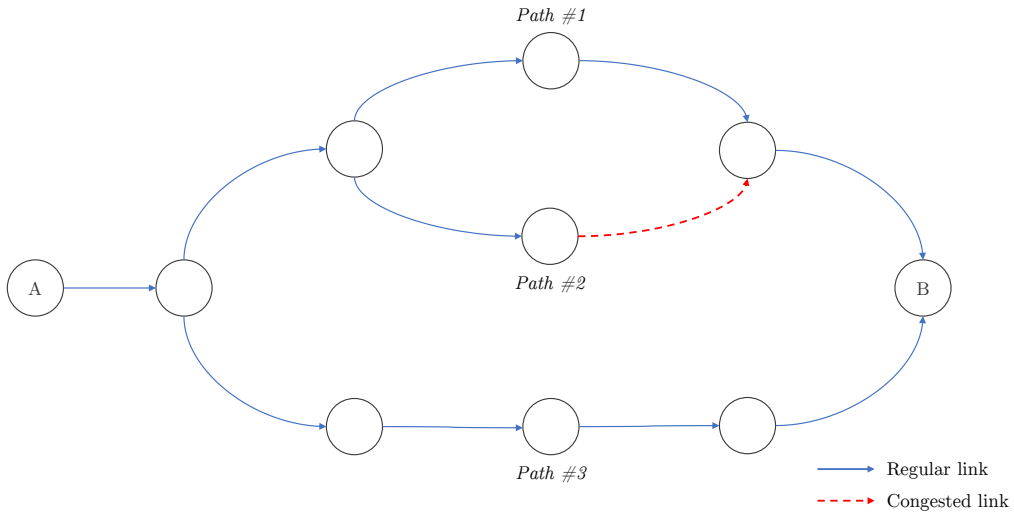


Figure 6.11: Illustration of a network topology with Equal-Cost-MultiPath routing. Packets going from A to B can follow three different paths. Traffic using path #2 will suffer from a greater latency due to a congested link.

Yet, classic tools like traceroute [49] or MTR [50] rely on ICMP mechanisms designed prior to the multipath-era, and can never ensure that they disclose all possible paths. This is a growing concern since multipath routes are frequently used in ISP [51] and datacenters [52] networks. More recent tools issued from academic research such as Paris Traceroute [53] add algorithms optimized for multipaths routes discovery on top of the well-known ICMP techniques, but even if they yield more accurate results, they still cannot deterministically guarantee that all routes have been discovered.

Using Segment Routing and `End.BPF`, we developed an intra-AS traceroute solution that is always capable to find all ECMP paths in an IPv6 network between two endhosts, at the expense of having on each ECMP router a SID bound to a specific OAM function designed for this task. We suggest a new type of SRv6 OAM request, enabling routers to indicate to a prober what are its ECMP routes for a queried destination. These requests are processed by our OAM SRv6 function implemented in BPF performing FIB lookups in kernel space to find the ECMP nexthops. A custom traceroute

tool is then designed and implemented, leveraging this new OAM feature, and its performances are assessed in a simulated ISP network topology.

6.3.1 `End.OAMP`: SRv6 function for dumping ECMP nexthops

At the time of writing, the draft on SRv6 OAM mechanisms [54] does not specify or suggest a method for discovering all ECMP paths between endpoints, although the issue is mentioned in section 2.3. We hence propose the following design. Whereas traceroute, MTR or Paris traceroute have to aggressively send probes with different properties (source or destination ports, flow label, ICMP sequence number, ...) to influence the output of the ECMP hashing at each hop, we suggest instead to add a feature on SRv6 routers to actively assist the tool performing the traceroute. Since `End.BPF` allows to add information into TLVs of an existing SRH, we leverage this feature to design the SRv6 function `End.OAMP` that inserts a TLV containing its ECMP nexthops for a given target destination address.

In our solution, next hops dump requests are sent to `End.OAMP` SIDs with a TLV (see figure A.6) containing the target for which the next hops are requested. This TLV can be attached to real SRv6 traffic, or to a synthetic probe.

When an IPv6 packet with SRH reaches an `End.OAMP` SID, the function verifies if a TLV containing an ECMP next hops dump request has been inserted. In that case, `End.OAMP` fetches all ECMP next hops for the target address contained in the request. A second TLV following the structure of figure A.7 and containing the next hops is subsequently added at the end of the SRH.[14] The packet is then forwarded to the next segment.

Implementation using `End.BPF`

This SRv6 function has been implemented in BPF using `End.BPF` in 75 SLOCs. It uses the `seg6_find_tlv` and `seg6_add_tlv` functions from libseg6 (which has been introduced in section 5.2) to, respectively, fetch the query TLV and insert the reply TLV.

The major prerequisite of our implementation is being capable of accessing the FIB in BPF. Until recently, BPF programs were unable to query the FIB, since accessing a kernel component requires specific helpers to be implemented, and none had been designed for this task. [55] However, a new helper `bpf_skb_fib_lookup`, which will also be released with Linux 4.18, has been introduced to perform FIB lookups, both for IPv4 and IPv6. It only returns one nexthop for each lookup and hence cannot be used to deterministically enumerate all possible nexthops. Instead, we forked this helper to create `bpf_ipv6_multipath_nh`, which performs a FIB lookup for a given IPv6 address and returns the associated next hops in a buffer. This buffer then forms the reply TLV once the 4 other fields have been inserted before the address of the first hop.

14. Due to the `HdrLen` field of the SRH which is only 1 byte long, the maximum number of next hops that can be inserted into a SRH without any other TLV, but the one containing the ECMP next hops dump request, is 126 minus the number of segments listed in the SRH. That said, this limit is not a practical problem, routers hardly reach more than 10 ECMP next hops. [53]

One particular issue arises when the FIB uses IPv6 link-local addresses as next hops. These addresses are only valid on a single link between two devices [56] and cannot be directly inserted in the reply TLV. `End.OAMP` has to translate them to a global unicast address belonging to the next hop router, but no mapping between these two types of addresses is available in the kernel, as it is usually built by the routing daemon, or by the network administrator when the router is statically configured. Our program relies on a BPF hash map to implement this mapping, and queries it when `bpf_ipv6_multipath_nh` returns a next hop in the `fe80::/10` prefix. Therefore, if link-local addresses are used in FIB, this hash map needs to be filled by the entity responsible for configuring the routing table.[15]

6.3.2 SRv6 multipath-aware traceroute algorithm

Leveraging `End.OAMP`, we then designed a multipath-aware traceroute tool for SRv6 networks named *segtrace*. It acts as a regular traceroute tool and tries to discover the closest routers and progressively continues until the destination has been reached, but with some key modifications.

First off, instead of building a naive list of hops (with possibly several routers per hop if there are ECMP routes), the algorithm uses a tree structure to keep track of the paths. A router with N ECMP routes will hence have its corresponding branch splitted in N sub-branches.

Once a node at distance K has been discovered, a SRv6 synthetic probe with the next hops dump request TLV is first sent to the corresponding `End.OAMP` SID of router. We make here the assumption that the network operators reserved across all routers the function identifier 8 for `End.OAMP`, even if it is not installed on the router (see figure 6.12). A second segment corresponding to the global unicast address of the prober is inserted to force the packet to come back. If the router R at distance K replies to this request, the next hops contained in the reply TLV are inserted in the tree as children of R. If there is no reply (e.g. because it is a legacy router, without OAM functionalities), the algorithm falls back to the legacy traceroute mechanism: three UDP probes with different source ports and a maximum number of IPv6 Hops K + 1 are sent.

Moreover, when UDP probes are sent, a SRH is injected with as segments all the intermediate routers discovered in the path being probed. This ensures that the ICMPv6 *Hops Limit Exceeded* reply originating from this probe are actually from routers belonging to the branch that we are inspecting. Otherwise the probe could freely go through any path between the endhosts and we could not map new discovered routers to a precise branch in the tree.

A crucial condition to ensure the optimal behavior of this algorithm is that all intermediate nodes with several ECMP routes must support `End.OAMP`. If this is not the case, the regular UDP probes cannot deterministically guarantee that all next hops will be found for a non-`End.OAMP` node. Otherwise, the final result is guaranteed to be exhaustive. Another prerequisite is that all routers within the network must support basic SRv6 routing on their global unicast addresses, otherwise even the UDP probes could not be forwarded, due to the presence of a SRH.

We implemented segtrace in Python. Its implementation requires 300 SLOC, is straightforward, and

15. We here neglect the risk of collision. The usage of a hash map over other data structures has been preferred because of its simplicity, but using an array instead is also possible.

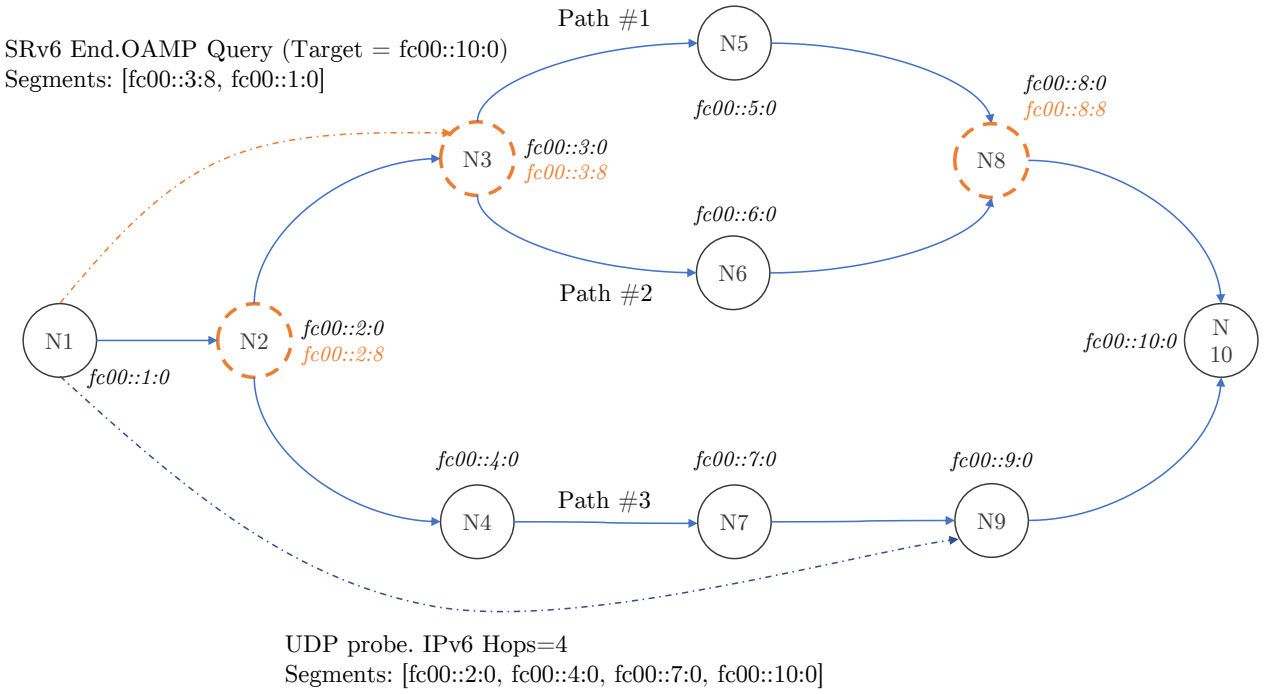


Figure 6.12: Illustration of the operation of our SRv6 traceroute. The prober is executed on N1. Nodes N2, N3 and N8 have an `End.OAMP` SID, whereas the other routers do not. Two examples of probes are shown, regular UDP and `End.OAMP`, with their corresponding segments. IPv6 addresses in orange are `End.OAMP` SIDs, the others in black are the regular global unicast addresses of the routers.

respects all the above-mentioned mechanisms. Injecting a SRH in the probes (both for `End.OAMP` and UDP) is performed through the system call `setsockopt` using the `IPV6_RTHDR` option. When a reply to an `End.OAMP` request arrives, the SRH containing the next hops is fetched via the ancillary data using `recvmsg`.

6.3.3 Evaluation in a simulated network

Our solution has been evaluated in a simulated network, virtualized using Linux's network namespaces.[16] This simulation deploys the topology of AT&T's backbone in the USA around 2008 (fetched from topology-zoo.org, see figure D.1). It contains 25 nodes and 57 links. These nodes are thus very interconnected, and among the 300 possible routes between all nodes, 165 have at least 2 equal-cost multipaths.

We first assessed the percentage of multipaths that our algorithm can discover, for 5 sets of router configurations. In each set, a fixed percentage of routers have been randomly configured with `End.OAMP` support. Figure 6.13 shows that the naive traceroute algorithm works well for short routes, but increasingly fails to discover all paths when the number of hops with possibly several ECMP next hops expands. Adding `End.OAMP` support to routers clearly enhances the results, even installing `End.OAMP` on only 50% of randomly chosen routers bumps the number of paths discovered

16. On each router, the IPv6 ECMP hashing policy has been set to use the standard 5-tuple. All links have been given a weight of 1. Rate limits for ICMPv6 messages have been disabled.

from 70% to 89% for routes with 4 hops.

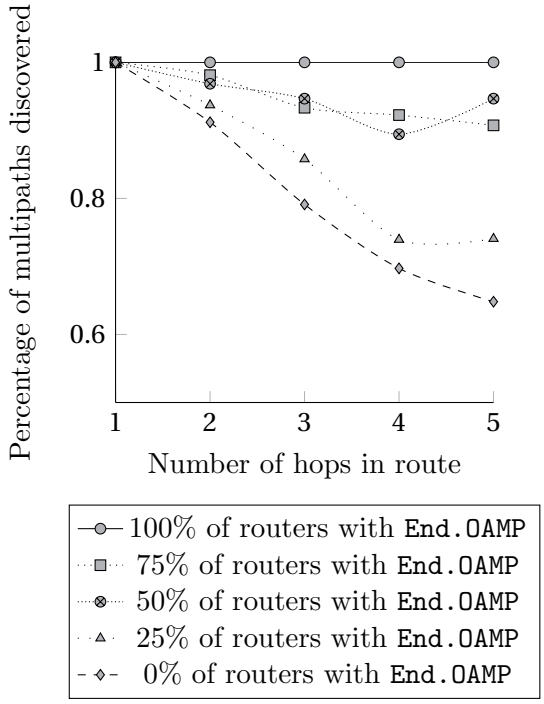


Figure 6.13: Accuracy of the multipaths discovery through 300 traceroutes for several percentages of `End.OAMP` support.

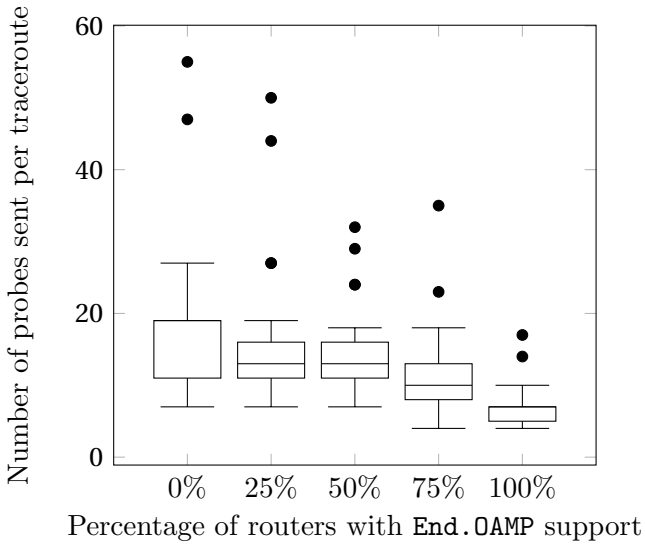


Figure 6.14: Evolution of the number of probes per traceroute for several percentages of `End.OAMP` support. Sampled from 24 traceroutes launched from the same node. Among the 24 routes, 11 have at least two paths. Average number of hops per route is 3,74.

Furthermore, since a single reply from a `End.OAMP` yields all possible ECMP next hops for a node, whereas otherwise three UDP probes are sent in the attempt to discover several next hops, the number of probes sent per traceroute is reduced by a third when all routers have been configured with `End.OAMP`, as indicated by figure 6.14. The two outliers present in all 5 sets correspond to two ECMP routes with 5 and 7 different paths and both going through 5 hops.

We also executed Paris Traceroute in this simulated network. However, when it was launched against a destination with more than 4 multipaths, it systematically hung and failed to deliver meaningful results. That said, [51] shows that Paris Traceroute is capable of detecting more than 4 multipaths. Since we could not explain the causes of this issue, we did not include these simulations in the above results.

6.3.4 Conclusion and discussion

This use-case demonstrates that BPF programs are capable of accessing information in various components in the kernel, provided that an adapted helper is available. More specifically, `End.OAMP` performs true IPv6 FIB lookups, which cannot be done in user space. A user space alternative implementation would have to fetch information from the RIB through netlink and try to mimic the inner functioning of the kernel IPv6 layer for FIB lookups, but this can become quite difficult when taking into account all possible parameters influencing lookups (routes metrics, ECMP hash-

ing algorithm being used, rules based on source and destination addresses or ports[17], ...). BPF hence enables new features leveraging kernel space mechanisms to be implemented, with a very low complexity.

Throughout `End.OAMP`, we also showed that information can easily be inserted into the SRH through TLVs and retrieved by other nodes in the network, either by routers or by endhosts. In the latter case, we showed that accessing information stored in a TLV is possible by accessing the ancillary data of a datagram received in user space. User space programs can also send SRHs with specially crafted TLVs using `setsockopt`. Sending and receiving TLVs from user space is a mechanism that could be useful for the design of other potential OAM use-cases.

Finally, an improved traceroute tool leveraging these mechanisms has been developed. Simulations have demonstrated that the more routers support `End.OAMP`, the more the final output converges to the exhaustive list of all multipaths possible for a given flow. We particularly think that this solution could be useful in datacenters, where administrators have full control of their network and could easily deploy `End.OAMP` on most of their routers. In this case, the deployment only consists of adding a simple `End.BPF` route with the corresponding BPF code, and possibly adding e.g. an extension to the routing daemon for filling the mapping between unicast global addresses and link-local ones, if the latter are used in the routing table.

17. Our implementation of `End.OAMP` is not capable of performing FIB lookups for a specific transport protocol and source or destination ports, but this functionality could easily be implemented by extending the ECMP next hops dump request TLV.

7 Conclusion

The main contribution of this thesis is `End.BPF`, an interface enabling to develop IPv6 Segment Routing network functions in Linux. This feature will be released in Linux 4.18, and will hence be available on millions of servers and personal computers around the world. This is by far the greatest achievement of our work, and we hope that this functionality will receive substantial adoption as SRv6 gets progressively deployed over the following years.

In addition, we developed an initial ecosystem for this interface, comprising a SRv6 unit testing framework, a general purpose library for SRv6 network functions and a generic user space daemon architecture.

Finally, we implemented three use-cases leveraging `End.BPF`. They allowed to assess the strengths and drawbacks of the interface. The analysis of their implementations and of their evaluations hinted some directions for future work regarding `End.BPF`.

Strengths, drawbacks and discussion

`End.BPF` can be analyzed from several viewpoints: performance, flexibility and complexity. The micro-benchmarks in 4.3 demonstrated that that the overhead due to the virtual machine is fairly minimal. Our delay monitoring solution showed that routers can instrument using `End.BPF` a fraction of the traffic for OAM purposes with virtually no impact on the forwarding performance. The link aggregation solution subsequently proved that modern CPEs are capable of leveraging BPF to forward traffic at the line rates of current access networks.

Although the three developed use-cases do not represent all possible features that could be implemented with `End.BPF`, they sketched an outline of the flexibility of this interface. The four SRv6 BPF helpers of 4.1 accommodated all the needs related to the SRv6 data plane (reading and adding TLVs, applying basic SRv6 actions, ...). We did not encounter limitations using the user space mechanisms of BPF (maps, perf events). However, two use-cases required to access information in kernel space that led us to develop additional BPF helpers. One major issue hence seems to be the amount and diversity of available helpers. That said, we are positive that this situation will improve. In one year, 16 new helpers have been implemented in the mainline kernel [55]. We expect the usage of BPF to substantially grow in the following years, sparking the implementation of more helpers.

Regarding the complexity viewpoint, all three use-cases were implemented in BPF on average with 100 SLOC, without taking into account libseg6 (230 SLOC). These numbers are a solid indication that the development of `End.BPF` programs is a fairly simple process, whereas creating new datapaths in the kernel would be a much more cumbersome task.

Open source contributions

Besides the acceptance of `End.BPF` in the mainline Linux kernel, this work was also at the origin of other open source contributions. Two bugs in the Linux SRv6 implementation have been identified and fixed [57, 58]. Support for the LWT BPF inside `pyroute2` has been implemented and sent upstream [59]. A minor bug in the SRv6 implementation of `scapy` has also been corrected [60]. `End.BPF` support for `iproute2` and `pyroute2` will be sent upstream in the month following the submission of this thesis.

Moreover, the code developed in this work has been released with an open source license. Future users can hence leverage the set of tools that we developed, and rely on the implemented use-cases as solid examples of what can be achieved using `End.BPF`.

Directions for future work

This thesis only laid the foundations of an interface for programmable IPv6 Segment Routing network functions in Linux. Further work could be carried out along the two following axes:

- **Improving** `End.BPF`: several ideas of amelioration have been presented in the conclusion of chapter 4. Although the infrastructure has been laid out, more features need to be implemented to allow a broader range of network functions to be developed with `End.BPF`
- **Extending the scope of SRv6 BPF**: the use-cases developed in this thesis focused only on three specific problems. We are positive that our work could pave the way to efficient solutions for problems in other domains of research related to SRv6, such as SDN and SFC.

We hope that the open source dimension of this work will incite both academic researchers and network operators to continue what has begun here. We demonstrated that associating the theoretical capabilities of SRv6 and the flexibility of an in-kernel virtual machine opened up new possibilities of network functions. This work brings the reality of programmable networks one step further, it is now up to the network operators to enter the dance.

Bibliography

[1] Clarence Filsfils, Nagendra Kumar Nainar, Carlos Pignataro, Juan Camilo Cardona, and Pierre Francois. The segment routing architecture. In *Global Communications Conference (GLOBECOM), 2015 IEEE*, pages 1–6. IEEE, 2015. (Referenced in pages 1, 3 and 4.)

[2] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. RFC 3031, RFC Editor, January 2001. URL http://www.rfc-editor.org/rfc/rfc3031.txt. http://www.rfc-editor.org/rfc/rfc3031.txt. (Referenced in page 3.)

[3] Bob Braden, Lixia Zhang, Steve Berson, Shai Herzog, and Sugih Jamin. Resource reservation protocol (rsvp) – version 1 functional specification. RFC 2205, RFC Editor, September 1997. URL http://www.rfc-editor.org/rfc/rfc2205.txt. http://www.rfc-editor.org/rfc/rfc2205.txt. (Referenced in page 3.)

[4] S. Yasukawa, A. Farrel, and O. Komolafe. An analysis of scaling issues in mpls-te core networks. RFC 5439, RFC Editor, February 2009. (Referenced in page 3.)

[5] Randeep Bhatia, Fang Hao, Murali Kodialam, and TV Lakshman. Optimized network traffic engineering using segment routing. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 657–665. IEEE, 2015. (Referenced in page 3.)

[6] Luca Davoli, Luca Veltri, Pier Luigi Ventre, Giuseppe Siracusano, and Stefano Salsano. Traffic engineering with segment routing: Sdn-based architectural design and open source implementation. In *Software Defined Networks (EWSDN), 2015 Fourth European Workshop on*, pages 111–112. IEEE, 2015. (Referenced in page 3.)

[7] Ahmed AbdelSalam, Francois Clad, Clarence Filsfils, Stefano Salsano, Giuseppe Siracusano, and Luca Veltri. Implementation of virtual network function chaining through segment routing in a linux-based nfv infrastructure. In *Network Softwarization (NetSoft), 2017 IEEE Conference on*, pages 1–5. IEEE, 2017. (Referenced in pages 3 and 12.)

[8] Clarence Filsfils et al. Srv6 network programming. Internet-Draft draft-filsfils-spring-srv6-network-programming-04, IETF Secretariat, March 2018. URL http://www.ietf.org/internet-drafts/draft-filsfils-spring-srv6-network-programming-04.txt. http://www.ietf.org/internet-drafts/draft-filsfils-spring-srv6-network-programming-04.txt. (Referenced in pages 3, 4, 6, 8, 11, 19, 23, 38 and 39.)

[9] Clarence Filsfils et al. Stefano Previdi. Ipv6 segment routing header (srh). Internet-Draft draft-ietf-6man-segment-routing-header-13, May 2018. URL http://www.ietf.org/internet-drafts/draft-ietf-6man-segment-routing-header-13.txt. http://www.ietf.org/internet-drafts/draft-ietf-6man-segment-routing-header-13.txt. (Referenced in pages 3, 4, 5 and 8.)

[10] Les Ginsberg Bruno Decraene Stephane Litkowski Clarence Filsfils, Stefano Previdi and Rob Shakir. Segment routing architecture. Internet-Draft draft-ietf-spring-segment-routing-14, IETF Secretariat, December 2017. URL http://www.ietf.org/internet-drafts/draft-ietf-spring-segment-routing-14.txt. http://www.ietf.org/internet-drafts/draft-ietf-spring-segment-routing-14.txt. (Referenced in pages 3 and 4.)

[11] S. Deering and R. Hinden. Internet protocol, version 6 (ipv6) specification. STD 86, RFC Editor, July 2017. (Referenced in pages 4 and 5.)

[12] David Lebrun and Olivier Bonaventure. Implementing ipv6 segment routing in the linux kernel. In *Applied Networking Research Workshop 2017*, July 2017. See https://irtf.org/anrw/2017/anrw17-final3.pdf. (Referenced in pages 9 and 10.)

[13] David Lebrun. *Reaping the Benefits of IPv6 Segment Routing*. PhD thesis, Ecole polytechnique de Louvain, Belgium, 2017. (Referenced in pages 9 and 11.)

[14] Vpp - segment routing for ipv6. https://wiki.fd.io/view/VPP/Segment_Routing_for_IPv6, 2018. [Online; accessed 8 June 2018]. (Referenced in page 11.)

[15] Yoann Desmouceaux, Pierre Pfister, Jérôme Tollet, Mark Townsley, and Thomas Clausen. Srlb: The power of choices in load balancing with segment routing. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 2011–2016. IEEE, 2017. (Referenced in pages 11 and 20.)

[16] David Lebrun Fabien Duchêne and Olivier Bonaventure. Srv6pipes: enabling in-network bytestream functions. In *IFIP Networking 2018*, 2018. (Referenced in page 12.)

[17] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 93, 1993. (Referenced in page 13.)

[18] Linux weekly news - a thorough introduction to ebpf. https://lwn.net/Articles/740157/, 2018. [Online; accessed 8 June 2018]. (Referenced in pages 13 and 15.)

[19] Daniel Borkmann. On getting tc classifier fully programmable with cls bpf. *Proceedings of netdev*, 2016. (Referenced in page 13.)

[20] Bpf and xdp reference guide. http://cilium.readthedocs.io/en/latest/bpf/, 2018. [Online; accessed 8 June 2018]. (Referenced in pages 13 and 16.)

[21] Linux kernel documentation - linux socket filtering aka berkeley packet filter (bpf). https://www.kernel.org/doc/Documentation/networking/filter.txt, 2018. [Online; accessed 8 June 2018]. (Referenced in page 14.)

[22] The llvm compiler infrastructure - project website. https://llvm.org/, 2018. [Online; accessed 9 June 2018]. (Referenced in page 14.)

[23] Brendan Gregg. perf examples. http://www.brendangregg.com/perf.html, 2018. [Online; accessed 8 June 2018]. (Referenced in page 16.)

[24] Thomas Graf. bpf: Bpf for lightweight tunnel encapsulation. https://lwn.net/Articles/708020/, 2016. [Online; accessed 8 June 2018]. (Referenced in page 16.)

[25] Thomas Graf. Github repository of the bcc project. https://github.com/iovisor/bcc, 2018. [Online; accessed 8 June 2018]. (Referenced in page 17.)

[26] Khaled Salah, Khalid Elbadawi, and Raouf Boutaba. Performance modeling and analysis of network firewalls. *IEEE Transactions on network and service management*, 9(1):12–21, 2012. (Referenced in page 19.)

[27] Ali et al. Performance measurement in segment routing networks with ipv6 data plane (srv6). Internet-Draft draft-ali-spring-srv6-pm-02, IETF Secretariat, February 2018. URL http://www.ietf.org/internet-drafts/draft-ali-spring-srv6-pm-02.txt. http://www.ietf.org/internet-drafts/draft-ali-spring-srv6-pm-02.txt. (Referenced in pages 20, 37, 38, 39, 40 and 41.)

[28] David Lebrun, Mathieu Jadin, François Clad, Clarence Filsfils, and Olivier Bonaventure. Software resolved networks: Rethinking enterprise networks with ipv6 segment routing. In *Proceedings of the Symposium on SDN Research*, page 6. ACM, 2018. (Referenced in page 20.)

[29] Clarence Filsfils et al. Ahmed Bashandy. Topology independent fast reroute using segment routing. Internet-Draft draft-bashandy-rtgwg-segment-routing-ti-lfa-04, IETF Secretariat, April 2018. URL http://www.ietf.org/internet-drafts/draft-bashandy-rtgwg-segment-routing-ti-lfa-04.txt. http://www.ietf.org/internet-drafts/draft-bashandy-rtgwg-segment-routing-ti-lfa-04.txt. (Referenced in page 20.)

[30] Jonathan Corbet. Driver porting: per-cpu variables. https://lwn.net/Articles/22911/, 2003. [Online; accessed 28 May 2018]. (Referenced in page 25.)

[31] Wireshark - display filter reference: Routing header for ipv6. https://www.wireshark.org/docs/dfref/i/ipv6.routing.html, 2018. [Online; accessed 3 June 2018]. (Referenced in page 31.)

[32] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao-keng Jerry Chu, Andreas Terzis, and Tom Herbert. packetdrill: Scriptable network stack testing, from sockets to packets. In *USENIX Annual Technical Conference*, pages 213–218, 2013. (Referenced in page 31.)

[33] pyroute2 - github repository. https://github.com/svinota/pyroute2, 2018. [Online; accessed 5 June 2018]. (Referenced in page 31.)

[34] scapy - packet crafting library for python2 and python3. https://scapy.net/, 2018. [Online; accessed 3 June 2018]. (Referenced in page 31.)

[35] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural guidelines for multipath tcp development. RFC 6182, RFC Editor, March 2011. URL http://www.rfc-editor.org/rfc/rfc6182.txt. http://www.rfc-editor.org/rfc/rfc6182.txt. (Referenced in pages 33 and 46.)

[36] D. Frost and S. Bryant. Packet loss and delay measurement for mpls networks. RFC 6374, RFC Editor, September 2011. (Referenced in pages 37 and 39.)

[37] Abhinav Pathak, Himabindu Pucha, Ying Zhang, Y Charlie Hu, and Z Morley Mao. A measurement study of internet delay asymmetry. In *International Conference on Passive and Active Network Measurement*, pages 182–191. Springer, 2008. (Referenced in page 37.)

[38] European Commission. State of the union 2016: Commission paves the way for more and better internet connectivity for all citizens and businesses. http://europa.eu/rapid/press-release_IP-16-3008_en.htm, 2016. [Online; accessed 16 May 2018]. (Referenced in page 43.)

[39] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 29–29. USENIX Association, 2012. (Referenced in page 43.)

[40] N. Leymann, C. Heidemann, M. Zhang, B. Sarikaya, and M. Cullen. Huawei's gre tunnel bonding protocol. RFC 8157, RFC Editor, May 2017. (Referenced in pages 43 and 45.)

[41] et al. C. Paasch, S. Barre. Multipath tcp in the linux kernel. Available from https://www.multipath-tcp.org, 2018. [Online; accessed 5 June 2018]. (Referenced in page 44.)

[42] Jiaxin Cao, Rui Xia, Pengkun Yang, Chuanxiong Guo, Guohan Lu, Lihua Yuan, Yixin Zheng, Haitao Wu, Yongqiang Xiong, and Dave Maltz. Per-packet load-balanced, low-latency routing for clos-based data center networks. 2013. (Referenced in page 44.)

[43] M. Allman, V. Paxson, and E. Blanton. Tcp congestion control. RFC 5681, RFC Editor, September 2009. URL http://www.rfc-editor.org/rfc/rfc5681.txt. http://www.rfc-editor.org/rfc/rfc5681.txt. (Referenced in page 45.)

[44] Wensong Zhang. Weighted round-robin scheduling. http://kb.linuxvirtualserver.org/wiki/Weighted_Round-Robin_Scheduling, 2005. [Online; accessed 16 May 2018]. (Referenced in page 47.)

[45] [iptables,1/2] extensions: libip6t_srh: support matching previous, next and last sid - netdev commit description. https://patchwork.ozlabs.org/patch/902882/, 2018. [Online; accessed 8 June 2018]. (Referenced in page 48.)

[46] Time based packet transmission - netdev patchset description. https://patchwork.ozlabs.org/cover/882342/, 2018. [Online; accessed 8 June 2018]. (Referenced in page 48.)

[47] Shubham Bansal. net-next commit: arm: ebpf jit compiler. https://patchwork.ozlabs.org/patch/804281/, 2018. [Online; accessed 5 June 2018]. (Referenced in page 50.)

[48] D. Thaler and C. Hopps. Multipath issues in unicast and multicast next-hop selection. RFC 2991, RFC Editor, November 2000. URL http://www.rfc-editor.org/rfc/rfc2991.txt. http://www.rfc-editor.org/rfc/rfc2991.txt. (Referenced in page 53.)

[49] FreeBSD. User manual of `traceroute`. https://www.freebsd.org/cgi/man.cgi?query=traceroute, 2015. [Online; accessed 1 June 2018]. (Referenced in page 53.)

[50] BitWizard B.V. Mtr. http://www.bitwizard.nl/mtr/, 2018. [Online; accessed 1 June 2018]. (Referenced in page 53.)

[51] Brice Augustin, Timur Friedman, and Renata Teixeira. Measuring load-balanced paths in the internet. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 149–160. ACM, 2007. (Referenced in pages 53 and 57.)

[52] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008. (Referenced in page 53.)

[53] Brice Augustin, Timur Friedman, and Renata Teixeira. Measuring multipath routing in the internet. *IEEE/ACM Transactions on Networking (TON)*, 19(3):830–840, 2011. (Referenced in pages 53 and 54.)

[54] Clarence Filsfils et al. Zafar Ali. Operations, administration, and maintenance (oam) in segment routing networks with ipv6 dataplane (srv6). Internet-Draft draft-spring-srv6-oam-00, IETF Secretariat, December 2017. URL http://www.ietf.org/internet-drafts/draft-spring-srv6-oam-00.txt. http://www.ietf.org/internet-drafts/draft-spring-srv6-oam-00.txt. (Referenced in page 54.)

[55] Bpf helpers - documentation. https://github.com/qmonnet/bpf-helpers/blob/master/out/bpf-helpers.rst, 2018. [Online; accessed 8 June 2018]. (Referenced in pages 54 and 59.)

[56] R. Hinden and S. Deering. Ip version 6 addressing architecture. RFC 4291, RFC Editor, February 2006. URL http://www.rfc-editor.org/rfc/rfc4291.txt. http://www.rfc-editor.org/rfc/rfc4291.txt. (Referenced in page 55.)

[57] Mathieu Xhonneux. [net] ipv6: sr: fix memory oob access in seg6_do_srh_encap/inline - netdev commit description. https://patchwork.ozlabs.org/patch/920389/, 2018. [Online; accessed 9 June 2018]. (Referenced in page 60.)

[58] Mathieu Xhonneux. [net,v2] ipv6: sr: fix tlvs not being copied using setsockopt - netdev commit description. https://patchwork.ozlabs.org/patch/858256/, 2018. [Online; accessed 9 June 2018]. (Referenced in page 60.)

[59] Mathieu Xhonneux. Add support of lwt bpf encap - pyroute2 pull request. https://github.com/svinota/pyroute2/pull/459, 2018. [Online; accessed 9 June 2018]. (Referenced in page 60.)

[60] Mathieu Xhonneux. Segment routing: fixing pseudo-header computation - scapy pull request. https://github.com/secdev/scapy/pull/858, 2018. [Online; accessed 9 June 2018]. (Referenced in page 60.)

[61] S. Bryant, S. Sivabalan, and S. Soni. Udp return path for packet loss and delay measurement for mpls networks. RFC 7876, RFC Editor, July 2016. (Referenced in page 68.)

Header and TLV structures

A

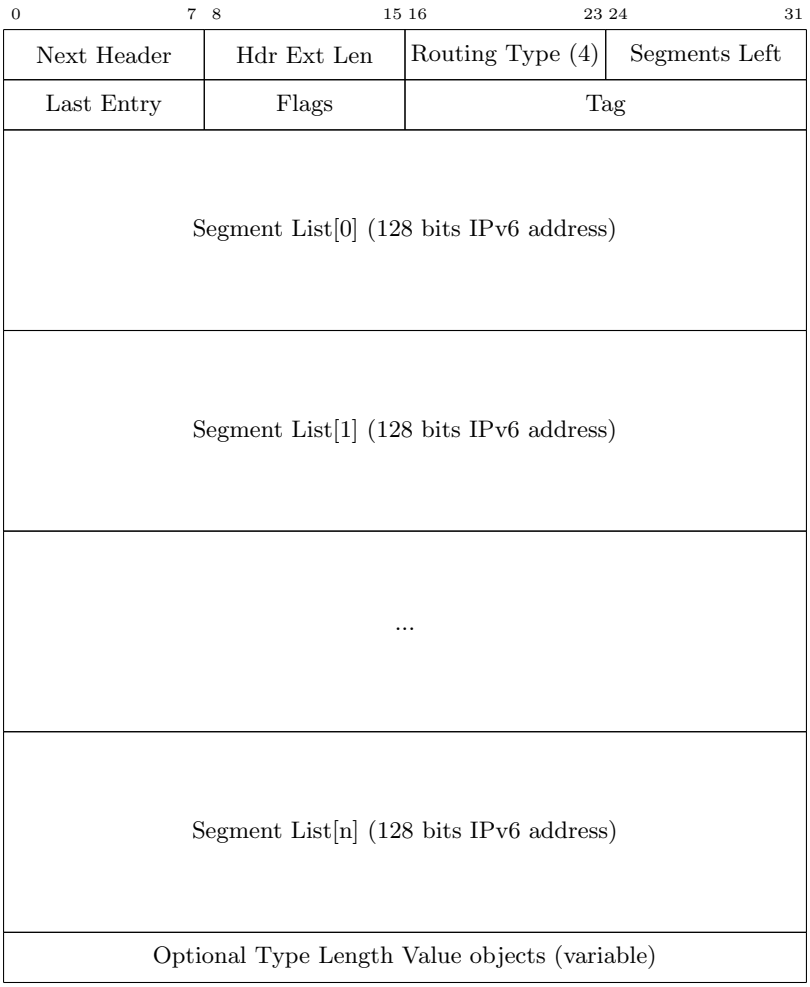


Figure A.1: Segment Routing Header

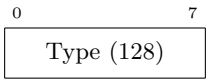


Figure A.2: PAD0 TLV

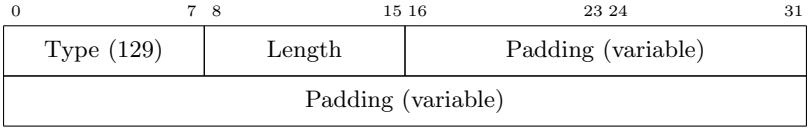


Figure A.3: PADN TLV

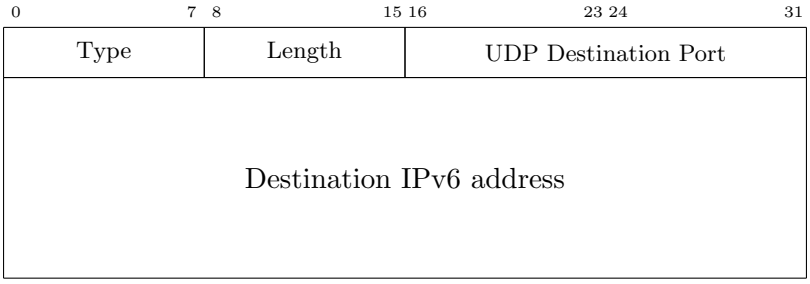


Figure A.4: Format of a IPv6 UDP Return Object [61].

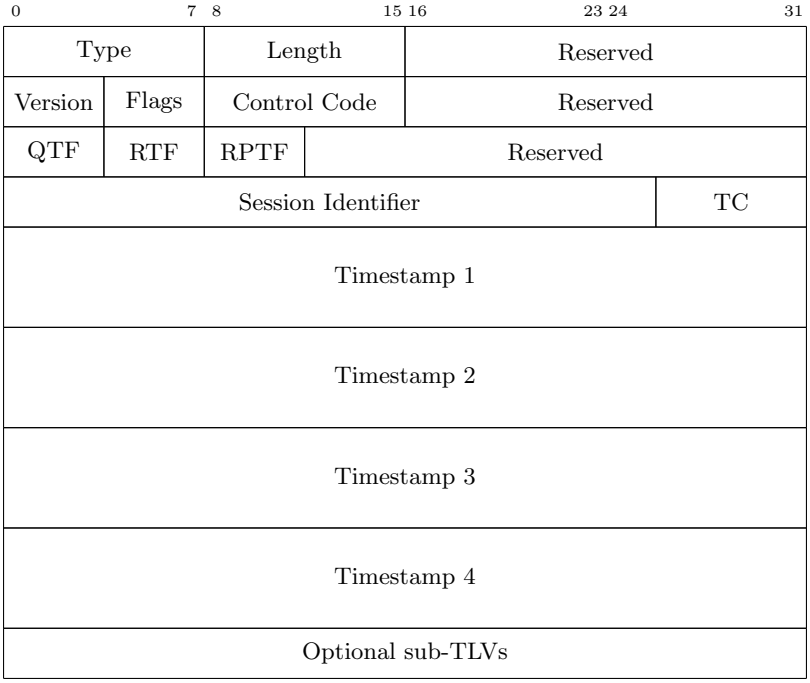


Figure A.5: Format of a Delay Measurement TLV.

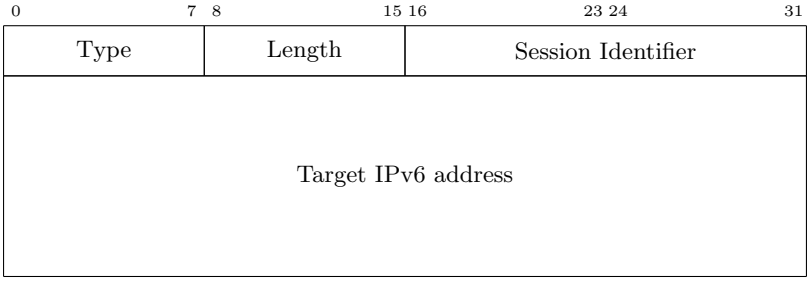


Figure A.6: TLV for requesting an ECMP next hops dump. *Session Identifier* has the same objective as for the DM TLV.

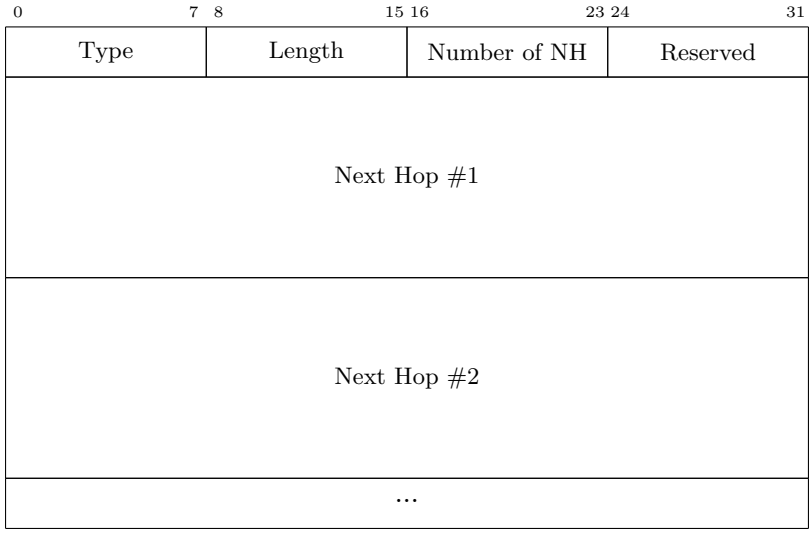


Figure A.7: TLV containing an ECMP nexthops dump.

Specifications of SRv6 network functions

B

Listing 14 Specifications of `End.X`.

```
1  IF NH=SRH and SL > 0
2      decrement SL
3      update the IPv6 DA with SRH[SL]
4      forward to layer-3 adjacency bound to the SID S
5   ELSE
6      drop the packet
```

Listing 15 Specifications of `End.T`.

```
1  IF NH=SRH and SL > 0
2      decrement SL
3      update the IPv6 DA with SRH[SL]
4      lookup the next segment in IPv6 table T associated with the SID
5      forward via the matched table entry
6  ELSE
7      drop the packet
```

Listing 16 Specifications of `End.B6`.

```
1  IF NH=SRH and SL > 0
2      do not decrement SL nor update the IPv6 DA with SRH[SL]
3      insert a new SRH
4      set the IPv6 DA to the first segment of the SRv6 Policy
5      forward according to the first segment of the SRv6 Policy
6  ELSE
7      drop the packet
```

Listing 17 Specifications of `End.B6.Encaps`.

```
1  IF NH=SRH and SL > 0
2      decrement SL and update the IPv6 DA with SRH[SL]
3      push an outer IPv6 header with its own SRH
4      set the outer IPv6 SA to A
5      set the outer IPv6 DA to the first segment of the SRv6 Policy
6      forward according to the first segment of the SRv6 Policy
7  ELSE
8      drop the packet
```

Listing 18 Specifications of `End.DT6`.

```
1  IF NH=SRH and SL > 0
2      drop the packet
3  ELSE IF ENH = 41
4      pop the (outer) IPv6 header and its extension headers
5      lookup the exposed inner IPv6 DA in IPv6 table T
6      forward via the matched table entry
7  ELSE
8      drop the packet
```

Specifications of SRv6 BPF helpers

C

Listing 19 Specifications of `bpf_lwt_push_encap`.

```
int bpf_lwt_push_encap(struct sk_buff *skb, u32 type, void *hdr, u32 len);
/* Description
 *  Encapsulate the packet associated to *skb* within a Layer 3
 *  protocol header. This header is provided in the buffer at
 *  address *hdr*, with *len* its size in bytes. *type* indicates
 *  the protocol of the header and can be one of:
 *
 *  **BPF_LWT_ENCAP_SEG6**
 *   IPv6 encapsulation with Segment Routing Header
 *   (**struct ipv6_sr_hdr**). *hdr* only contains the SRH,
 *   the IPv6 header is computed by the kernel.
 *  **BPF_LWT_ENCAP_SEG6_INLINE**
 *   Only works if *skb* contains an IPv6 packet. Insert a
 *   Segment Routing Header (struct ipv6_sr_hdr) inside
 *   the IPv6 header.
 *
 *   A call to this helper is susceptible to change the underlaying
 *   packet buffer. Therefore, at load time, all checks on pointers
 *   previously done by the verifier are invalidated and must be
 *   performed again, if the helper is used in combination with
 *   direct packet access.
 * Return
 *   0 on success, or a negative error in case of failure.
 */
```

Listing 20 Specifications of `bpf_lwt_seg6_store_bytes`.

```
int bpf_lwt_seg6_store_bytes(struct sk_buff *skb, u32 offset, const void *from
    , u32 len);
/* Description
 *  Store *len* bytes from address *from* into the packet
 *  associated to *skb*, at *offset*. Only the flags, tag and TLVs
 *  inside the outermost IPv6 Segment Routing Header can be
 *  modified through this helper.
 *
 *   A call to this helper is susceptible to change the underlaying
 *   packet buffer. Therefore, at load time, all checks on pointers
 *   previously done by the verifier are invalidated and must be
 *   performed again, if the helper is used in combination with
 *   direct packet access.
 * Return
 *   0 on success, or a negative error in case of failure.
 */
```

Listing 21 Specifications of `bpf_lwt_seg6_adjust_srh`.

```
int bpf_lwt_seg6_adjust_srh(struct sk_buff *skb, u32 offset, s32 delta);
/* Description
 *  Adjust the size allocated to TLVs in the outermost IPv6
 *  Segment Routing Header contained in the packet associated to
 *  *skb*, at position *offset* by *delta* bytes. Only offsets
 *  after the segments are accepted. *delta* can be as well
 *  positive (growing) as negative (shrinking).
 *
 *   A call to this helper is susceptible to change the underlaying
 *   packet buffer. Therefore, at load time, all checks on pointers
 *   previously done by the verifier are invalidated and must be
 *   performed again, if the helper is used in combination with
 *   direct packet access.
 * Return
 *   0 on success, or a negative error in case of failure.
 */
```

Listing 22 Specifications of `bpf_lwt_seg6_actions`.

```
int bpf_lwt_seg6_action(struct sk_buff *skb, u32 action, void *param, u32
    param_len);
/* Description
 *  Apply an IPv6 Segment Routing action of type *action* to the
 *  packet associated to *skb*. Each action takes a parameter
 *  contained at address *param*, and of length *param_len* bytes.
 *  *action* can be one of:
 *
 *  **SEG6_LOCAL_ACTION_END_X**
 *   End.X action: Endpoint with Layer-3 cross-connect.
 *   Type of *param*: *struct in6_addr*.
 *  **SEG6_LOCAL_ACTION_END_T**
 *   End.T action: Endpoint with specific IPv6 table lookup.
 *   Type of *param*: *int*.
 *  **SEG6_LOCAL_ACTION_END_B6**
 *   End.B6 action: Endpoint bound to an SRv6 policy.
 *   Type of param: *struct ipv6_sr_hdr*.
 *  **SEG6_LOCAL_ACTION_END_B6_ENCAP**
 *   End.B6.Encap action: Endpoint bound to an SRv6
 *   encapsulation policy.
 *   Type of param: *struct ipv6_sr_hdr*.
 *
 *   A call to this helper is susceptible to change the underlaying
 *   packet buffer. Therefore, at load time, all checks on pointers
 *   previously done by the verifier are invalidated and must be
 *   performed again, if the helper is used in combination with
 *   direct packet access.
 * Return
 *   0 on success, or a negative error in case of failure.
 */
```

AT&T network topology

D

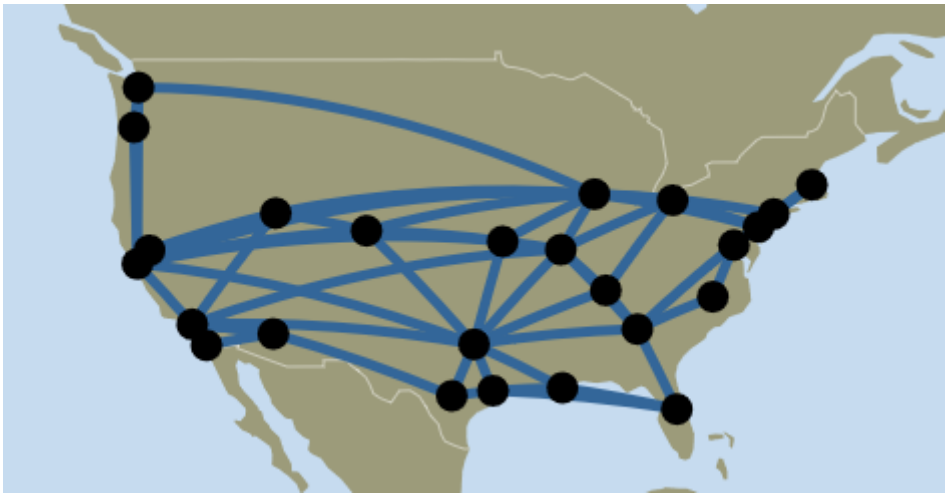


Figure D.1: Topology of AT&T's backbone in the US, circa 2008.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve www.uclouvain.be/epl