Implementing IPv6 Segment Routing in the Linux Kernel

David Lebrun, Olivier Bonaventure
ICTEAM, Université catholique de Louvain
Louvain-La-Neuve – Belgium
firstname.lastname@uclouvain.be

ABSTRACT

IPv6 Segment Routing is a major IPv6 extension that provides a modern version of source routing that is currently being developed within the Internet Engineering Task Force (IETF). We propose the first open-source implementation of IPv6 Segment Routing in the Linux kernel. We first describe it in details and explain how it can be used on both endhosts and routers. We then evaluate and compare its performance with plain IPv6 packet forwarding in a lab environment. Our measurements indicate that the performance penalty of inserting IPv6 Segment Routing Headers or encapsulating packets is limited to less than 15%. On the other hand, the optional HMAC security feature of IPv6 Segment Routing is costly in a pure software implementation. Since our implementation has been included in the official Linux 4.10 kernel, we expect that it will be extended by other researchers for new use cases.

1. INTRODUCTION

Segment Routing (SR) is a modern source routing architecture that is being developed within the Internet Engineering Task Force [1, 2]. While traditional IP routing uses destination-based hop-by-hop forwarding, Segment Routing can forward packets along non-shortest paths towards their destination by specifying a list of detours or waypoints called *segments*. Packets are forwarded along the shortest path from the source to the first segment, then through the shortest path from the first segment to the second segment, and so on. Segment Routing defines two main types of segments representing topological instructions. A *node* segment is used to steer packets through a specific network node. An *adjacency* segment allows to steer packets through a particular link.

The Segment Routing architecture has been instantiated in two different dataplanes: MPLS [3] and IPv6 [4]. The MPLS variant of SR is already implemented by network vendors and deployed by operators. Several researchers have proposed new traffic engineering solutions that leverage the unique characteristics of SR [5, 6, 7, 8, 9]. Other researchers have proposed improved monitoring solutions [10]

The IPv6 SR dataplane is younger and its specification became stable in early 2017 [4]. We argue that the inherent features of the IPv6 flavor of Segment Routing (SRv6) open the doors to a large, yet mostly unexplored, research area enabling the endhosts to actively participate in the selection of the paths followed by their packets.

SRv6 is realised through an IPv6 extension header, the routing header (RH). This header was already defined in the IPv6 protocol specification [11]. The IPv6 Segment Routing Header (SRH) is defined as an extension of the routing header [4]. The SRH contains a list of segments, encoded as IPv6 addresses (usually globally routable and announced by an underlying IGP such as OSPF or IS-IS). A segment can represent a topological instruction (node or link traversal) or any operator-defined instruction (*e.g.*, virtual function). An SRH can be used to steer packets through paths with given properties (*e.g.*, bandwidth or latency) and through various network functions (*e.g.*, firewalling). The list of segments present in the SRH thus specifies the network policy that applies to the packet. Each SRH contains at least a list of segments [4] and a `Segments Left` pointer that references the current active segment (a value of 0 refers to the last segment). In addition, an SRH can optionally include extended information encoded as Type-Length-Value (TLV) fields. One example is the optional HMAC TLV, used for authenticity and integrity checks.

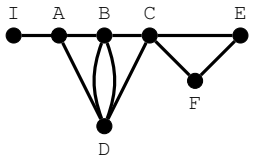
When a router must impose an SRH on a forwarded packet, the packet is encapsulated in an outer IPv6 header containing the SRH. The original packet is left unmodified as the payload. The router is called the *SR ingress node*. The destination address of the outer IPv6 header is set to the first segment of the list and the packet is forwarded to the corresponding node following the shortest path. This node (called a *segment endpoint*) then processes the packet by updating the destination address to the next segment. The last segment of the list is the *SR egress node*. It decapsulates the inner packet and forwards it to its original destination. Figure 1 shows an illustration of path steering with Segment Routing.

Another SRH insertion technique is direct (or *inline*) insertion. In this case, the original packet is not encapsulated. Rather, the SRH is directly inserted immediately after the IPv6 header. This method yields less overhead than encapsulation, but is more susceptible to disruptions in case of network errors. For example, an ICMP generated for such a packet will reach the original source of the packet, which does not know about the inserted SRH. However, such a technique might be used within an SRv6 domain to, *e.g.*, decouple the IPv6 encapsulation and the SRH insertion.

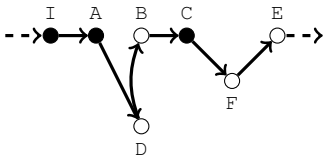
A particular feature of SRv6 is that it can be used in the network up to the endhosts, as they already support (or are meant to support) IPv6, as opposed to MPLS which is only used in core networks. Such feature enables the endhosts to actively participate in

ANRW '17, July 15, 2017, Prague, Czech Republic

DOI: http://dx.doi.org/10.1145/3106328.3106329



(a) Segment Routing domain.



(b) Forwarding path for segments `node(D)`, `adj(D, B, left)`, `node(F)`, `node(E)` imposed by SR ingress node `I`. Hollow nodes are segment endpoints.

Figure 1: Illustration of Segment Routing topological instructions.

the management and policing of their network traffic, with the help of an SDN-like controller. The first step in realising such an architecture would be to enable the support of SRv6 in the endhosts. Thus, we implemented SRv6 in the Linux kernel.

This paper is organised as follows. We first describe our open-source implementation of IPv6 Segment Routing in the Linux kernel in Section 2. We then analyse its performance in Section 3.

2. IMPLEMENTATION

Open-source implementations have played an important role on the development of all major Internet protocols. When the implementation is developed in parallel with the protocol specification, it helps to validate key design choices and also triggers new ideas. Our first implementation of IPv6 Segment Routing was written in April 2014. During the last three years, we have refined and improved it to track the evolution of the protocol specification. We describe in this section the latest version of this implementation that has been merged in the official Linux tree and is part of Linux 4.10 released in February 2017 [12, 13]. To the extent of our knowledge, this is the only open-source implementation of Segment Routing that supports both endhosts and router functionalities. The `fd.io` project has recently announced another implementation that focuses on router functionalities[1]. In this section, we first briefly explain the basic principles of networking in the Linux kernel. Then, we describe the key components of our SRv6 implementation and the supporting userspace tools.

2.1 Networking in the Linux Kernel

The Linux networking subsystem is very complex, comprising more than 700,000 lines of code, without counting the drivers. Our IPv6 SR implementation mainly interacts with the IPv6 packet processing and the routing engine codes. Internally, packets are represented as socket buffers, or `skb`'s. An `skb` is a kernel structure of type `struct sk_buff` that represents a network packet, with metadata and payload. The metadata includes the ingress interface, checksum, header offsets, and other layer-specific data. The actual packet data is most often stored in a contiguous memory area, called the `skb` header. When all the packet data is contained in this area, the `skb` is said to be *linear*. The header is divided in three zones: headroom, packet data, and tailroom. Headroom and tailroom are space resp. before and after the packet data. They enable to prepend and append data to the packet. As packets are constructed from bottom to top, data is often prepended. If there is not enough head or tail room, the header is reallocated to a larger memory area. The `skb` also contains various offsets to easily access to different parts of the packet, such as the network header, transport header, inner headers (in case of encapsulation), etc.

[1] See http://www.segment-routing.net/open-software/vpp/

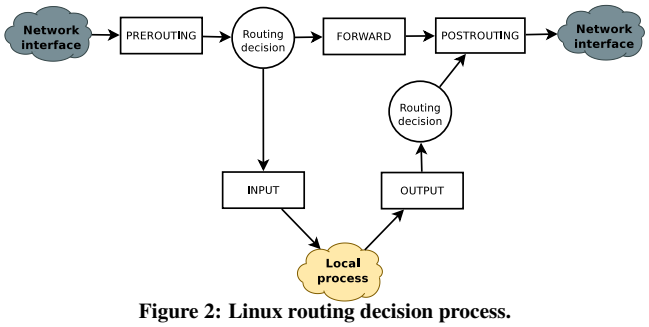


Figure 2: Linux routing decision process.

2.1.1 Packet processing

The networking subsystem of the Linux kernel is divided into several layers. The lowest level is the network driver, which is closest to the hardware. When a packet arrives at the network interface card (NIC), it is copied into main memory and handed to the network driver. The driver transforms the raw packet into a socket buffer. Once the `skb` is filled, it is handed over to the upper layer, e.g. IPv4 or IPv6. The driver may concatenate several `skb`'s using Generic Receive Offloading (GRO). GRO is a software mechanism that enables network drivers to group similar contiguous packets before handing them over to the upper layer, reducing the number of calls.

Upon reception of an IP packet, the kernel decides whether its should be delivered locally, or forwarded to another network node. Figure 2 provides an overview of the routing Linux decision process. Each packet goes through several processing stages. Consider an IPv6 packet that was just delivered by the network driver. It enters the `ipv6_rcv()` function that corresponds to the `PREROUTING` stage. This function parses the IPv6 header, then decides whether the packet is intended to the local host or must be forwarded elsewhere. To realise this, it looks up the packet destination in the IPv6 routing table entries [2]. If the packet must be forwarded, it is handed to the `ip6_forward()` function, corresponding to the `FORWARD` stage. This function selects the next hop, decrements the hop limit, etc. Then, it enters the `ip6_output()` function, corresponding to the `POSTROUTING` stage. In this stage, the packet is ready to be transmitted and is handed over to the lower layers, ultimately being transmitted by the NIC. On the other hand, if the packet is to be locally delivered (*i.e.*, the destination address matches a local address), it enters the `ip6_input()` function, corresponding to the `INPUT` stage. This function iteratively processes each extension header in the IPv6 header chain until it reaches a final payload (*e.g.*, TCP, UDP), which is processed by its own input function.

When an application sends packets, they are built from bottom to top. First, the transport header is pushed above the payload. Then, optional IPv6 extension headers are pushed on top of the transport header thanks to the `ipv6_push_nfrag_opts()` function. Finally, the IPv6 header is pushed on top and the correspond-

[2] In practice, other parameters can be used in the routing decision process, such as the source address. For the sake of simplicity, we only consider destination addresses.

Listing 1: SRH structure.

```
struct ipv6_sr_hdr {
        __u8    nexthdr;
        __u8    hdrlen;
        __u8    type;
        __u8    segments_left;
        __u8    first_segment;
        __u8    flags;
        __u16   reserved;

        struct in6_addr segments[0];
};
```

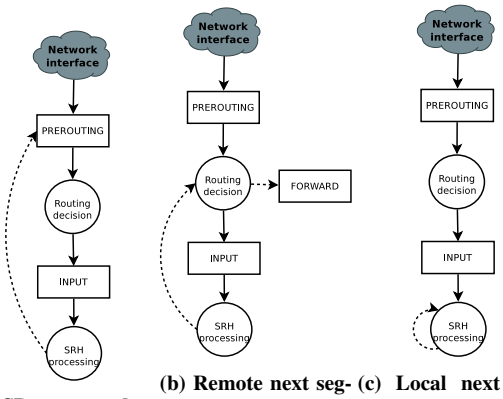
ing `skb` enters the `OUTPUT` stage through, *e.g.*, the `ip6_xmit()` function for TCP or the `ip6_local_out()` function for other transport protocols. The routing decision is performed and the `skb` then enters the `POSTROUTING` stage and is transmitted to the network.

2.2 IPv6 SR data plane

The core of our IPv6 SR implementation is the Segment Routing Header processing capability. It enables a Linux node to act as both a segment endpoint and an SR egress node. When a segment endpoint receives an IPv6 packet containing an SRH, the destination address of the packet is local to the segment endpoint. To process this packet, we add the `ipv6_srh_rcv()` function the `INPUT` stage. This function is called whenever the IPv6 input function encounters an SRH in the header chain. We use the C structure shown in Listing 1 to hold an SRH.

The `ipv6_srh_rcv()` function performs several operations. First, it checks that the node is allowed to act as a segment endpoint for SR-enabled packets coming from the ingress interface (`skb->dev`). This policy is configured through the per-interface `seg6_enabled sysctl` boolean parameter. If this boolean is set to false, then the `skb` is discarded. Otherwise, the processing continues. The packet then goes through an optional HMAC validation. Our implementation supports SHA-1 and SHA-256 based HMAC. The behavior to adopt when encountering HMAC-enabled packets is controlled through the per-interface `seg6_hmac_require sysctl`. This parameter can take three different values: (i) a value of -1 means that the node accepts all SR packets, regardless of the status (absent/present, valid/invalid) of the HMAC TLV, (ii) a value of 0 means that the node must accept packets that contain an SRH and either do not include the HMAC TLV or a valid HMAC TLV, and (iii) a value of 1 means that only the SR packets that include a valid HMAC can be processed.

Once those preliminary checks have been performed, the function handles two main cases: `Segments Left` being non-zero, and `Segments Left` being equal to zero. Let us first consider the latter case, where the node acts as an SR egress node. As the node is the last segment of the path, it must inspect the inner header to decide the fate of the packet. If the next header is another IPv6 extension header or a transport protocol (*e.g.*, TCP, UDP), then the header chain processing continues as normal, and the `skb` is eventually delivered to the corresponding local process. Conversely, if the original packet was encapsulated by an SR ingress node, then the next header would typically be an IPv6 header (*i.e.*, IPv6-in-IPv6 encapsulation). However, nothing forbids, *e.g.*, the encapsulation of an IPv4 packet within the outer IPv6 header and the SRH. As such, it would make sense to allow the kernel to continue the



(a) SR egress node. **(b) Remote next segment.** **(c) Local next segment.**

Figure 3: Possible codepaths for SRH processing.

default processing of the next header, as for the non-encapsulated case. However, the kernel handles IP headers differently, depending on whether it is the outermost header (the first header processed on the ingress interface, not counting the possible Ethernet header) or an inner, encapsulated header. When the kernel encounters an encapsulated inner header, it attempts to find an existing stateful tunnel interface, corresponding to the source and destination of the outer header. As no such interface exists, the `skb` is dropped. To avoid this issue, we bypass the default processing and reinject the inner packet in the ingress interface. The side-effect is that the function explicitly checks for an inner IP header. As of Linux 4.10, only IPv6-in-IPv6 encapsulation is supported, meaning that, *e.g.*, IPv4 packets cannot be transported in an IPv6 and SRH encapsulation.

When the Linux node is an intermediate segment endpoint (*i.e.*, `Segments Left` > 0), it must forward the packet towards the next segment. To realise this, the `ipv6_srh_rcv()` function decrements the number of segments left and updates the destination address of the packet to the next segment. Afterwards, a routing decision is applied to the `skb` thanks to the `ip6_route_input()` function. If the next segment is local to the node, then the `skb` is looped back to the beginning of the SRH receive function, after decrementing and checking the hop limit. This is an implementation choice that enables to skip a redundant re-entry into the `ip6_input()` function. In the future, this behavior might be controlled by a user-defined parameter. This parameter would enable to choose between fast-path loop-back and slow-path re-entry. The rationale is that the fast re-entry skips the `INPUT` netfilter hook, which may be needed in some usecases. If the next segment is non local, then the `skb` enters the `FORWARD` stage.

Figure 3 summarises the flow of an IPv6 SR packet through the networking subsystem. The codepath of the `skb` to the SRH processing function is shown with plain arrows. The dashed arrows show the codepath of the `skb` immediately after the SRH processing. Figure 3a shows the reinjection of the decapsulated `skb` at the interface level. Figure 3b shows the `skb` being forwarded to the next segment. Figure 3c shows the `skb` being looped back within the SRH processing function to handle a local next segment.

2.3 IPv6 SR control plane

An SR router must be able to add and remove SRH in IPv6 packets. We first describe how our implementation supports the insertion of an SRH inside an IPv6 packet[4, 14]. Our implementation leverages the *lightweight tunnels* (LWTs). LWTs are a technique to implement interfaceless, virtual tunnels. The idea

Listing 2: `iproute2` command to insert an IPv6 SR encapsulation route.

```
ip -6 route add fc42::/64 encap seg6 mode encap
   segs fc00::1,2001:db8::1,fc10::7 dev eth0
```

of lightweight tunnels is the following. Each route in the kernel routing table is associated with two function pointers, `input` and `output`. Those pointers are initialized at the creation of the route. For an IPv6 route, the `output` function pointer references the `ip6_output()` function, that transmits the `skb` to the egress interface. The `input` function pointer depends on whether the route is local (packets matching this route must be delivered to a local process) or non-local (packets must be forwarded to a next hop). If the route is local, the function pointer references the `ip6_input()` function. Otherwise, it is `ip6_forward()` that is referenced. Lightweight tunnels allow to override those function pointers with custom functions. To implement a lightweight tunnel, one needs to define specific input and/or output functions. Then, each route created to specifically use this LWT will have its `input` and/or `output` function pointers reference the custom functions. Per-tunnel stateful data (*tunnel state*) is also stored inside the route. This technique has the advantage of using the existing routing table, thus avoiding the need to define a custom data structure for packet matching. It is also highly customizable, enabling differentiated treatment for forwarded packets and for locally generated packets. Lightweight tunnels are configured from userspace through the `rtnetlink` protocol, which is commonly used by the `iproute2` tool to configure the routing tables.

Consequently, we implemented SRH insertion using the lightweight tunnels. When such SRH insertion is associated to a route, the entire SRH is passed to the kernel through `rtnetlink`. After checking the consistency of the SRH, it is stored as the tunnel state of the route. Our implementation also stores an optional parameter stating whether the SRH should be directly inserted or encapsulated. Finally, a `dst_cache` entry stores the routing entry associated with the first segment of the SRH, enabling the route lookup for the first segment to be performed only once. Both the `input` and `output` function pointers of the route ultimately call the `seg6_do_srh()` function. This function effectively inserts the SRH on the `skb`. The direct insertion function (`seg6_do_srh_inline()`) simply inserts the SRH between the IPv6 header and the rest of the packet. The encapsulation function (`seg6_do_srh_encap()`) needs to provision a new IPv6 header. The traffic class and the flowlabel are copied from the inner IPv6 header. In the future, this behaviour will likely be made configurable, *e.g.*, by letting the kernel compute the outer flowlabel based on the inner packet's 5-tuple. Such a behavior would ensure proper ECMP support when the inner flowlabel is null or unrelated to the packet's flow. The hop limit is also copied from the inner IPv6 header. The destination address is obviously set to the address of the first segment. The source address is selected according to a namespace-wide configuration parameter. In the future, this parameter could be directly attached to the routes for a finer-grained configuration. Once an `skb` is augmented with an SRH, it is forwarded to the first segment according to the normal kernel routing mechanisms.

We also modified the `iproute2` userspace tool to insert, modify and read routes that use the SRH lightweight tunnels extension. Figure 2 shows an example of a route insertion command. This route matches packets whose destination belongs to the `fc42::/64` prefix and inserts an SRH on these packets (`encap seg6`). The original packet is encapsulated in an outer IPv6 header (`mode encap`). To directly insert the SRH in the original packet, one can use `mode inline`. The list of segments is specified as a comma-separated list of IPv6 addresses. Finally, a non-loopback device must be specified.

Listing 3: Sample code to define a per-socket SRH.

```
struct ipv6_sr_hdr *srh;
int srh_len;

srh_len = build_srh(&srh);
fd = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
setsockopt(fd, IPPROTO_IPV6, IPV6_RTHDR, srh, srh_len);
```

The routes configured by `iproute2` are namespace-wide. To support a finer-grained control of the SRH insertion, we also implemented a per-socket interface through the `setsockopt()` system call. This enables applications to specify the SRH to be inserted on a socket level. When the kernel builds a packet for a local application, it calls `ipv6_push_nfrag_opts()` before pushing the top IPv6 header. If an SRH is attached to the socket, the function calls `ipv6_push_rthdr4()` that effectively pushes the SRH on the packet. The last segment of the SRH is set to the original destination. The first segment is returned, to be set as the actual destination of the packet. Listing 3 shows how a C application can define an SRH for a TCP socket.

2.4 HMAC

To support the HMAC TLV, we leverage the kernel crypto API, which implements various cryptographic functions. This crypto API requires the allocation of algorithm descriptors. To prevent allocating one such descriptor per packet, we leverage global per-CPU allocations. We pre-allocate algorithm descriptors at the initialization of the IPv6 SR and reuse them in the packet processing function. We support the SHA-1 and SHA-256 hashing algorithms.

The `seg6_hmac_compute()` function handles most of the HMAC computation. It takes as input an SRH, an IPv6 source address and computes the corresponding HMAC. The function does not use locks as it uses per-CPU buffers. However, it needs to disable preemption, to avoid being switched to another CPU in the middle of a computation. The `seg6_hmac_compute()` function can be called by several parts of our implementation. One of them is in the `ipv6_srh_rcv()` function, to verify the validity of an SR-enabled packet containing an HMAC TLV. Another one is the `seg6_do_srh()` function, that augments packets with an SRH including an HMAC TLV. Finally, an HMAC may also be computed for a packet generated through a local socket that has received an SRH through the `setsockopt()` system call.

To define an IPv6 SR encapsulation route with an HMAC, `iproute2` provides the `hmac` parameter that takes a key ID as argument. The SRH sent to the kernel contains a template HMAC TLV with the key ID set and the HMAC value zeroed. The same construction applies for a per-socket SRH. Those key IDs are configured through the `genetlink` protocol.

3. PERFORMANCE EVALUATION

To measure the performance of our IPv6 SR implementation in the Linux kernel, we ran several measurement campaigns on Xeon servers.

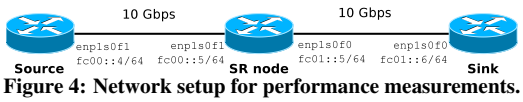


Figure 4: Network setup for performance measurements.

Our measurement setup uses three identical machines, equipped with Intel Xeon X3440 processors with 4 cores and 8 threads clocked at 2.53 GHz. Each server has 16 GB of RAM and two Intel 82599 10 Gbps network interface cards. One of them acts as a router and the two others act as sources and destinations as shown in Figure 4. The router acts as an SR node where we configure different operations. The source has a route to prefix `fc01::/64` via `fc00::5` and the sink has a route to `fc00::/64` via `fc01::5`.

Each server runs Linux kernel 4.11-rc3[3]. We compiled this kernel with all SR-related options enabled. The preemption model is voluntary and the clock ticks are set to 100 Hz periodic. We have disabled GRO and GSO on the network interfaces, as well as all hardware transmit and receive offloading features. On such servers, Linux usually configures each network interface with one queue per CPU (*i.e.*, 8 queues in our setup) and the IRQ associated with each queue is handled by the corresponding CPU. In order to observe performance bottlenecks, we change this setting to force all the queues of a network interface to be served by a single CPU. The other interface parameters are left at their default setting.

We use `pktgen` [15] to generate IPv6 packets. It is directly included in the kernel. As such, packets are directly handed over to the network driver, without further preprocessing. Consequently, `pktgen` is much faster than userspace counterparts such as `iperf3`. A drawback is that `pktgen` is not able to support reliable protocols such as TCP. It only sends raw UDP packets. However, this is sufficient for our tests since our objective is to evaluate the additional processing overhead imposed by IPv6 Segment Routing in comparison with regular IPv6 processing.

3.1 Measurements

To evaluate the performance impact of our IPv6 Segment Routing implementation, we carried out four measurement campaigns. We first measure **Plain**, regular IPv6 forwarding, without any SRH as a baseline. Our second campaign, **Encap**, evaluates the performance impact of encapsulating each IPv6 packet inside an outer packet that includes an SRH that contains one segment. Our third campaign, **Inline**, evaluates the impact of inserting an SRH inside each packet as proposed in [14]. Finally, we analyse during the **HMAC** campaign the cost of validating the HMAC TLV.

Unless stated otherwise, for each measurement we run 100 batches of five millions IPv6 packets having a length of 64 bytes, which is the minimum length supported by `pktgen`. Each packet includes an IPv6 header (40 bytes), a UDP header (8 bytes) with source and destination ports set to 9 (*discard* service) and a UDP payload of 16 bytes. The source IPv6 address is set to `fc00::44` (source server in Figure 4) and the destination address is set to `fc01::66` (the sink). When the SR node inserts an SRH with one segment, the SRH contains the segment `fc01::6`.

Our first measurement analysed the forwarding capacity of our test server with plain IPv6 packets. In the lab shown in Figure 4, we managed to forward packets at 1,165 Kpps. Figure 5 shows that this forwarding rate was pretty stable and did not vary significantly from one measurement to another. In the same setup, we compare encapsulation and direct insertion. They reached only 776 Kpps

[3]The exact version is at commit `add641e7dee31b36aee83412c29e39dd1f5e0c9c` in net-next.

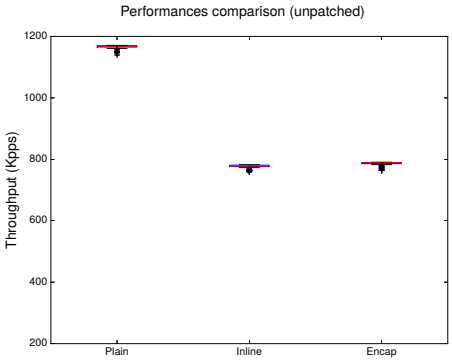


Figure 5: Initial performances for encapsulation and insertion.

for encapsulation and 784 Kpps for insertion. While the results reported in Figure 5 are not catastrophic, they are roughly one third lower than plain IPv6 forwarding performance on the same hardware.

We analysed the reason for this lower performance with the `perf` tool on the SR node while transmitting packets and compared the functions that were active during plain IPv6 forwarding and SRH insertion. This analysis pinpointed two kernel functions that were consuming more CPU time: `fib6_lookup()` and `__slab_free()`. The former is the IPv6 route lookup function. We realised that it was called once too often per SR-processed packet. This is because we implemented the `dst_cache` mechanism for locally generated packets (in `seg6_output()`) but not for forwarded packets (in `seg6_input()`). We fixed this problem by implementing the caching mechanism in both functions. The root cause of the `__slab_free()` increased usage was a little more difficult to determine. This function is called by `kfree()` (freeing kernel memory) when the data to free was not allocated by the same CPU as the one attempting to free the memory. In this case, a slowpath is taken which calls the `__slab_free()` function, itself taking a spinlock. This was likely because we were performing some memory allocation in the SRH insertion codepath. Indeed, `pskb_expand_head()` was called for each packet to increase the size of the `skb`'s headroom by the length of the IPv6 header and SRH that we were pushing onto the packet. However, the `skb` was originally allocated by the CPU that handled the hardware interrupt generated by the network card when receiving the packet. Unfortunately, the CPU that handles the IPv6 SR processing functions is not necessarily the same as the one that allocated the `skb`. As such, when the initial CPU attempts to free the `skb`, part of its data has been reallocated by another CPU, and thus the freeing process must take the slowpath through `__slab_free()`. To fix this issue, we leverage the already available headroom inside `skb`'s. We replace the `pskb_expand_head()` calls by `skb_cow_head()`, which reallocates the `skb` header only if the headroom is not large enough. After applying the two patches[4], we performed a second measurement campaign whose results are shown in Figure 6. The performance gain is clearly noticeable, with direct insertion and encapsulation reaching average forwarding rates of resp. 1,019 Kpps and 1,001 Kpps (standard deviation resp. 11.1 Kpps and 7.8 Kpps).

We then analysed the effect of the packet size on the performance of the SRH insertion. For this campaign, we replaced the 64-byte

[4]These two patches have been sent on the netdev mailing list. They have been accepted by the Linux kernel maintainers and will be part of Linux 4.12.

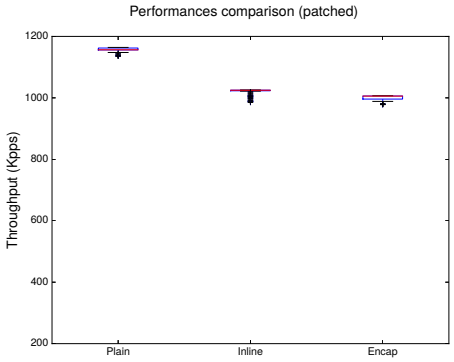


Figure 6: Performance for encapsulation and insertion after optimizations.

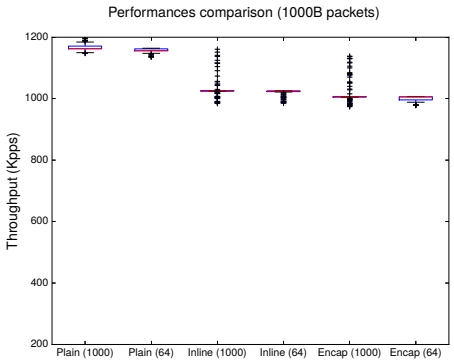


Figure 7: Performance for encapsulation and insertion using small and large packets.

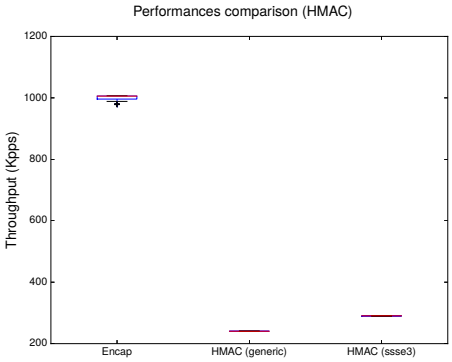


Figure 8: Performances for encapsulation with and without HMAC.

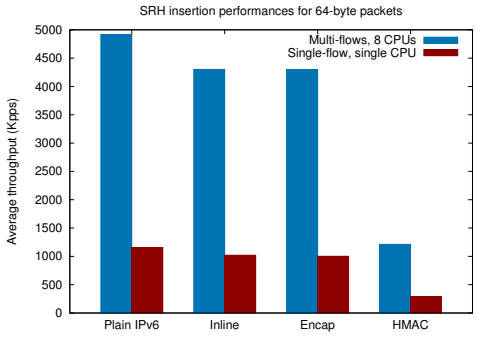


Figure 9: Performances summary for SRH encapsulation.

packets by 1000-bytes packets. Figure 7 shows the results for the **Plain**, **Inline** and **Encap** measurements, with both packet sizes. We observe that the performance with 1000-bytes packets is slightly better, especially for peak throughput. The average is almost the same for 1000-bytes and 64-bytes packets, but the standard deviation is three times higher for large packets. In any case, our measurements show that the packet size does not significantly impact the performance of inserting an SRH in each packet.

With our latest modifications, the performance penalty of using SRH insertion or encapsulation is very small. Among the other features of IPv6 Segment Routing, the validation of the HMAC TLV is likely the one that will have the highest performance impact, even if there are many environments where it will not be used [4]. Our fourth measurement campaign analyses the impact of SRH encapsulation with an HMAC, using the SHA-256 algorithm. We measured two implementations of SHA-256: a generic, purely software one and a partially hardware-offloaded one that leverages the `ssse3` instruction set. Our measurements with the `ssse3` implementation are reported in Figure 8. The performance penalty of using the HMAC TLV is huge. Using the generic implementation of SHA-256, we only reached 240 Kpps. The `ssse3`-augmented version reaches 290 Kpps in average, which is a long way from the 1,001 Kpps baseline for encapsulation without HMAC. The standard deviation is below 1 Kpps for both SHA-256 implementations. These results indicate that additional hardware offload capabilities will be required if the HMAC TLV is required for a specific deployment as noted in [4].

Finally, we measured the performances of parallel SRH processing. For each measurement campaign, we generated a random destination address for each packet, to simulate multiple flows. We enabled the Receive Side Scaling feature on the SR node, to uniformly distribute the flows over all the 8 CPUs. The results show that we reach almost 4.5 Mpps for SRH insertion. Figure 9 shows a summary of the performance measurements.

4. CONCLUSION

IPv6 Segment Routing is a major IPv6 extension that provides a modern version of source routing. It is gaining a growing interest within the IETF, with major network operators and vendors contributing to its design, as well as researchers. In this paper, we have described and evaluated the performance of our implementation of SRv6 in the Linux kernel. We support the SRv6 functions that are required on endhosts and routers. To the extent of our knowledge, this is the first implementation that supports these two use cases. The other existing implementations focus on router platforms.

On endhosts, our implementation allows applications to attach an SRH on a per-flow basis. On routers, it can forward SR-enabled packets, but also impose an SRH on packets matching a specific route, using direct insertion or encapsulation. We performed measurements on Xeon servers to assess the performance of our implementation. These results show that SRH processing reaches about 90% of the plain IPv6 forwarding performance.

Since our implementation has been included in the official Linux 4.10 kernel, we expect that it will be extended by other researchers for new use cases such as [16] and hope that it will become the reference implementation of IPv6 Segment Routing.

Acknowledgements

This work was partially supported by a Cisco grant and by the ARC grant 13/18-054 (ARC-SDN) from Communauté française de Belgique. We would like to thank Clarence Filsfils, Stefano Previdi and Eric Vyncke for fruitful discussions on IPv6 Segment Routing. We also thank Eric Dumazet, Tom Herbert and David Miller whose comments have helped us to improve the kernel code.

5. REFERENCES

[1] Clarence Filsfils et al. Segment Routing Architecture. Internet-Draft draft-ietf-spring-segment-routing-08, Internet Engineering Task Force, May 2016. Work in Progress.

[2] Clarence Filsfils et al. The segment routing architecture. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2015.

[3] Clarence Filsfils et al. Segment Routing with MPLS data plane. Internet-Draft draft-ietf-spring-segment-routing-mpls-07, Internet Engineering Task Force, February 2017. Work in Progress.

[4] Stefano Previdi, Clarence Filsfils, Brian Field, Ida Leung, J. Linkova, Ebben Aries, Tomoya Kosugi, Eric Vyncke, David Lebrun, John Leddy, Kamran Raza, daniel.voyer@bell.ca, daniel.bernier@bell.ca, Satoru Matsushima, Dirk Steinberg, and Robert Raszuk. IPv6 Segment Routing Header (SRH). Internet-Draft draft-ietf-6man-segment-routing-header-06, Internet Engineering Task Force, March 2017. Work in Progress.

[5] Renaud Hartert, Stefano Vissicchio, et al. A declarative and expressive approach to control forwarding paths in carrier-grade networks. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 15–28. ACM, 2015.

[6] Fang Hao, Murali Kodialam, and TV Lakshman. Optimizing restoration with segment routing. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pages 1–9. IEEE, 2016.

[7] François Aubry, David Lebrun, Yves Deville, and Olivier Bonaventure. Traffic duplication through segmentable disjoint paths. In *IFIP Networking Conference (IFIP Networking), 2015*, pages 1–9. IEEE, 2015.

[8] 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.

[9] Stefano Salsano et al. PMSR: Poor Man's Segment Routing, a minimalistic approach to Segment Routing and a Traffic Engineering use case. In *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*, pages 598–604. IEEE, 2016.

[10] Francois Aubry, David Lebrun, Stefano Vissicchio, Minh Thanh Khong, Yves Deville, and Olivier Bonaventure. Scmon: Leveraging segment routing to improve network monitoring. In *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016*, pages 1–9, 2016.

[11] Steve Deering. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, December 1998.

[12] Linux Torvalds. Linux 4.10 networking merge commit. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ce38aa9cbed3d109355b0169b520362c409c0541, December 2016.

[13] Linux community. Linux 4.10 ChangeLog. https://kernelnewbies.org/Linux_4.10, February 2017.

[14] D. Boyer et al. Insertion of IPv6 Segment Routing Headers in a Controlled Domain. Internet draft draft-voyer-6man-extension-header-insertion-00, work in progress, March 2017.

[15] Robert Olsson. Pktgen the linux packet generator. In *Proceedings of the Linux Symposium, Ottawa, Canada*, volume 2, pages 11–24, 2005.

[16] C. Filsfils et al. SRv6 Network Programming. Internet draft, draft-filsfils-spring-srv6-network-programming-00, work in progress, March 2017.