Mitigating the Double-Reallocation Issue for IPv6 Lightweight Tunnel Encapsulations

Anonymous Author(s)

Abstract

Lightweight Tunnels (LWTs) in the Linux kernel enable efficient per-route tunneling and are widely used by protocols such as **I**n Situ **O**perations, **A**dministration, and **M**aintenance (IOAM), **S**egment **R**outing over IPV6 (SRv6), and **R**outing **P**rotocol for **L**ow-Power and Lossy Networks (RPL). However, a performance issue was detected in their implementations, where a double-reallocation of socket buffers occurs under specific conditions, leading to significant throughput degradation. This paper investigates the root cause of the issue, which depends on the architecture of the Central Processing Unit (CPU) and the Network Interface Card (NIC). We propose a patch for the Linux kernel to fix this problem, replacing the double-reallocation with a single, efficient one. Performance evaluation demonstrates that the patch eliminates the inefficiency, improving forwarding rates by up to 28.8% for affected protocols.

Keywords

Linux, Networking, Performance, IOAM, SRv6, RPL

Introduction

In the Linux kernel, IP tunnels are commonly implemented using Lightweight Tunnels (LWTs) [17]. LWTs enable the creation of tunnels on a per-route basis, allowing custom encapsulation mechanisms to be defined through user-specified functions based on the packet destination. Various kernel networking units leverage LWTs, including:

- **I**n Situ **O**perations, **A**dministration, and **M**aintenance (IOAM) [4], a network telemetry protocol carrying operational data from devices (*e.g.*, routers or switches), such as latency and buffer size, directly into packet headers as they traverse the network.
- **S**egment **R**outing over IPV6 (SRv6) [8], a source routing protocol encoding the routing path into IPV6 headers, enabling simplified traffic engineering and flexible network programmability.
- **R**outing **P**rotocol for **L**ow-Power and Lossy Networks (RPL) [1], a routing protocol designed for resource-constrained networks, particularly in Internet-of-Things (IoT) building optimized multi-hop tree-like structures for reliable communication in dynamic and lossy environments.
- **M**ulti**P**rotocol **L**abel **S**witching (MPLS) [20], a technology designed to speedup forwarding decisions (through exact label matching instead of longest prefix matching on IP addresses) but is nowadays mainly deployed for providing IGP/BGP scalability and virtual private network (VPN) services [16].
- **I**dentifier **L**ocator **A**ddressing for IPV6 (ILA) [12], a way to differentiate between location and identity of a network node.
- Virtual `xfrm` interfaces [15] used for route-based VPN tunnels.

When running some IOAM performance evaluations, we detected an issue in its implementation. The problem lied in the LWT encapsulation of IOAM and occurs exclusively in this mode of operation. Such a behavior was particularly hard to understand since it occurred only under some conditions, that depend on the architecture of the Central Processing Unit (CPU) and the Network Interface Card (NIC). For those specific cases, and during the LWT encapsulation in packets, the kernel performs a double-reallocation of socket buffers instead of a single one, leading to obvious performance degradation. After some investigations, we found similar code patterns causing the exact same issue in the SRv6 and RPL implementations. To quantify the impact on performance, we measured the packet forwarding rate on a recent kernel version and observed up to 28.8% degradation.

This paper aims at addressing the aforementioned issue by proposing a Linux kernel patch that eliminates this inefficiency. Our patch replaces the double-reallocation with a single, optimized one. In addition, to validate its effectiveness, we conduct a performance evaluation of the three affected protocols, comparing the original implementation with our patched version, demonstrating its ability to improve forwarding efficiency.

Background

This paper focuses on a bug in the Lightweight Tunnel (LWT) implementation of IOAM, SRv6, and RPL protocols. In this section, we provide some background to readers by describing the aforementioned protocols, the structure of a socket buffer, and the role of the `skb_cow_head()` function in the Linux kernel.

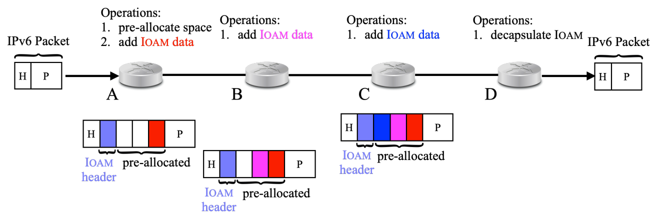


Figure 1: Generic example of IOAM data insertion. "H" corresponds to the IPV6 header, while "P" is the IPV6 payload.

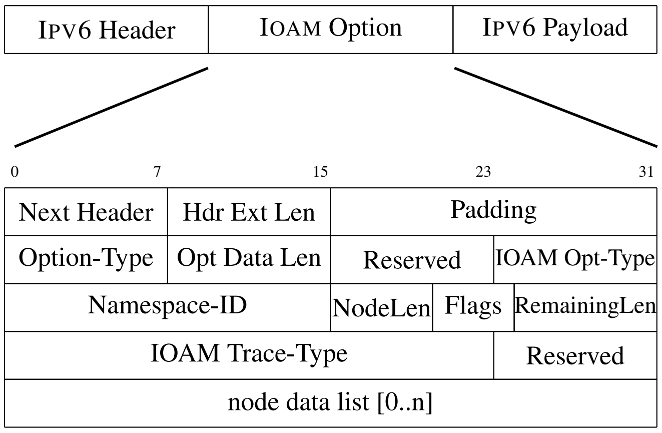


Figure 2: Hop-by-Hop with an IOAM PTO Option [4].

In Situ Operations, Administration, and Maintenance (IOAM)

In Situ **O**perations, **A**dministration, and **M**aintenance (IOAM) [4] provides a way to collect telemetry data from network devices (*e.g.*, routers or switches) along a path inside a given IOAM domain (e.g., within an ISP or Data Center Networks). "In Situ" refers to the fact that IOAM does not use dedicated packets for carrying the telemetry data. Rather, it relies on existing user traffic. IOAM telemetry can be encapsulated in a variety of protocols, including IPV6 [2] or **N**etwork **S**ervice **H**eader (NSH) [3]. In this paper, we only consider the IPV6 encapsulation.

An IOAM domain contains three types of nodes. First, the *encapsulating* nodes, located at the entry points (or INGRESS) of the domain, are responsible for appending the IPV6 Extension Header required for storing IOAM telemetry (node A in Fig. 1). Second, the *transit* nodes add telemetry data inside the existing IPV6 Extension Header (nodes B and C in Fig. 1). Finally, the *decapsulating* nodes, positioned at the exit points (or EGRESS) of the domain, are responsible for removing the IPV6 Extension Header (node D in Fig. 1). The format of the IPV6 Extension Header to carry IOAM telemetry is represented in Fig. 2.

There are two possibilities for encapsulating the IOAM data in IPV6. If the packet source is the encapsulating node, the IOAM data fields can be embedded in the existing IPV6 header. Otherwise, the IOAM data must be added inside an additional IPV6 header, since in-transit modification of existing headers is forbidden [6]. This approach leads to the creation of an IPV6-in-IPV6 tunnel across the IOAM domain. IOAM telemetry inside IPV6 can be encapsulated in either a Hop-by-Hop Extension Header, which is processed by every node, or a Destination Extension Header, which is only processed by the destination [2].

While IOAM defines several possibilities for adding telemetry data to packets [4, 18], this paper only considers the **P**re-allocated **T**race **O**ption (PTO) in which encapsulating nodes allocate the required space inside the IPV6 Extension Header in advance. PTO is illustrated in Fig. 1, where node A pre-allocates the required space for adding IOAM data.

In the kernel, the following encapsulation modes are supported for IPV6 IOAM:

- *Inline* mode: Directly added into the original packet header, which allows telemetry data to be included without creating an additional encapsulating header.
- *Encapsulation* (or *Tunnel*) mode: The original packet is encapsulated within a new IPV6 outer header, which contains telemetry data. This creates an IPV6-in-IPV6 tunnel.
- *Automatic* mode: The kernel either applies the *Inline* or *Tunnel* mode depending on the source of packets (*i.e.*, *Inline* mode if it is the source, *Tunnel* mode otherwise).

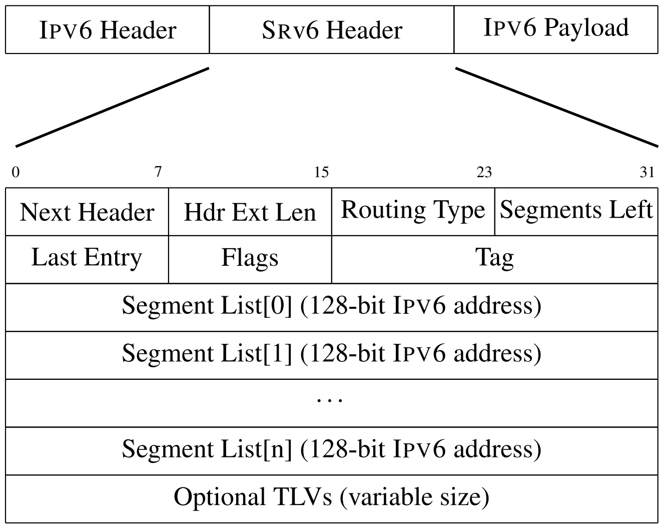

IPV6 Header
SRV6 Header
IPV6 Payload
0
7
15
23
31
Next Header
Hdr Ext Len
Routing Type
Segments Left
Last Entry
Flags
Tag
Segment List[0] (128-bit IPV6 address)
Segment List[1] (128-bit IPV6 address)
...
Segment List[n] (128-bit IPV6 address)
Optional TLVs (variable size)

Figure 3: SRV6 Header [9].

Segment Routing over IPV6 (SRV6)

In a nutshell, Segment Routing [8] is a loose source routing paradigm based on an ordered list of *segments* (*i.e.*, one or more instructions). Each segment can enforce a topological requirement (*e.g.*, pass through a node or an interface) or a service requirement (*e.g.*, execute an operation on the packet). Over the years, Segment Routing has found a suitable usage for many use cases such as network monitoring, traffic engineering, or failure recovery [19], among others. Two forwarding plane implementations are proposed for Segment Routing: Segment Routing over MPLS– SR-MPLS [7]

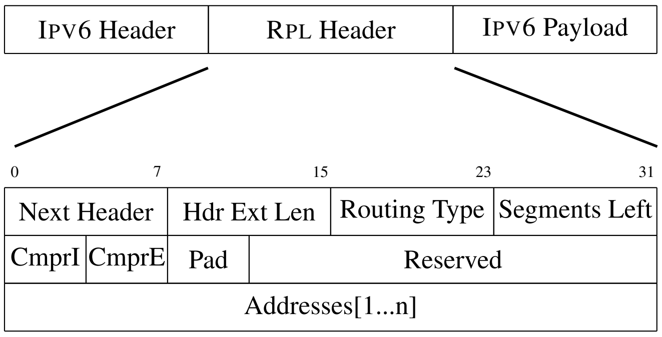


Figure 4: RPL Header [13].

and Segment Routing over IPV6– SRv6 [10]. SR-MPLS requires no change to the MPLS forwarding plane, while SRv6 is based on an Extension Header called SRv6 Header. In this paper, we only consider SRv6.

Segment Routing defines multiple types of segments, but the two most common are *node segments* and *adjacency segments*. A *node segment* represents the IGP least cost path between any router and a specified prefix. These segments can contain one or multiple IGP hops and have domain-wide significance. In normal Segment Routing operations, every Segment Routing router will announce a *node segment* for itself, allowing any router in the domain to know how to reach it. An *adjacency segment* represents an IGP adjacency between two routers and will cause a packet to traverse that specified link. These segments only have local significance. In normal Segment Routing operations, every Segment Routing router advertises an *adjacency segment* for each of its links.

Each segment is identified by a unique number, a *Segment IDentifier* (SID) implemented as a 128-bit IPV6 address in SRv6, enabling deployments over non-MPLS networks or areas without MPLS, such as data centers. This implementation simplifies deployments as it only requires advertising IPV6 prefixes. SIDs are encoded within the Routing Extension Header known as SRv6 Header [9] (see Fig. 3).

In the fashion of IOAM, the Linux kernel supports both the *Inline* and *Tunnel* encapsulation modes for SRv6. Additionally, SRv6 features a reduced encapsulation variant (referred to as “Red”) to optimize header size. It also offers a Layer-2 (“L2”) encapsulation variant, where the received frame is encapsulated within the IPV6 packet.

Routing Protocol for Low-Power and Lossy Networks (RPL)

RPL [1] is a routing protocol for wireless networks, especially suitable for resources-constrained networks such as Internet-of-Things (IoT). It is a Distance Vector Routing Protocol that creates a tree-like routing topology called the Destination Oriented Directed Acyclic Graph (DODAG), rooted towards one or more nodes called the root node or sink node.

For downward routing in non-storing mode (one of RPL’s modes of operation), RPL uses a Source Routing header to deliver datagrams, as shown in Fig. 4. Just like SRv6, this header contains a list of addresses that the packet must traverse to reach its destination. It comes with a compression mechanism, called Address Abbreviation, designed to reduce the size of the Routing header. If all nodes in the path share a common prefix, only the unique interface identifiers of each node are included. Therefore, the RPL header introduces the `CmprI`, `CmprE`, and `Pad` fields to allow compaction of the `Addresses[1...n]` vector when all entries share the same prefix as the IPV6 destination address of the packet. The `CmprI` and `CmprE` fields indicate the number of prefix octets that are shared with the IPV6 destination address of the packet. The shared prefix octets are not carried within the RPL header. The `Pad` field indicates the number of unused octets that are used for padding.

On the contrary to IOAM and SRv6, the Linux kernel only supports the *Inline* encapsulation mode for RPL.

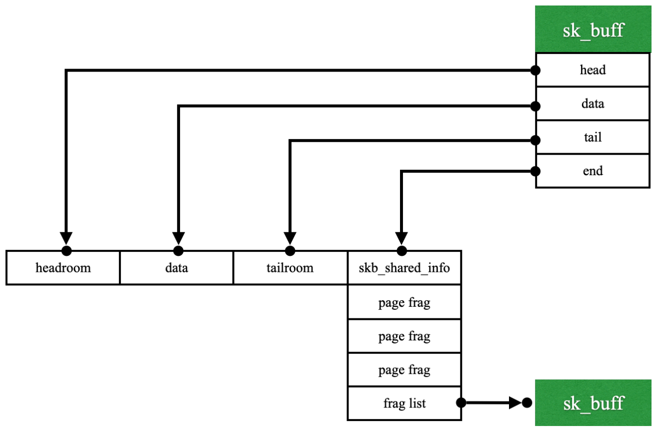


Figure 5: Basic `sk_buff` diagram [14].

Linux Kernel Socket Buffer

A socket buffer (`sk_buff`) in the Linux kernel is a core networking structure used to represent network packets. It acts as a container for the packet’s metadata, enabling efficient processing, routing, and management of network traffic.

Fig. 5 illustrates the layout of the `sk_buff` structure. It consists of several key components:

- `headroom`: Free space available for prepending headers.
- `data`: This area holds the headers and payload.
- `tailroom`: Free space available for adding trailing data if necessary.
- `skb_shared_info`: This structure holds an array of pointers to read-only data.

This design allows the kernel to modify headers or add encapsulations while minimizing memory reallocations.

The `skb_cow_head()` function

Listing 1 shows the definition of the `skb_cow_head()` function. Its purpose is to make sure that a socket buffer (*i.e.*, `skb` argument) has at least the desired headroom size (*i.e.*, `headroom` argument). It is usually used when one only

```
static inline int skb_cow_head(
   struct sk_buff *skb,
   unsigned int headroom
);
```

Listing 1: Definition of the `skb_cow_head()` function.

```
1    err = skb_cow_head(skb, hdrlen + skb->mac_len);
2    // add the new header ("hdrlen" bytes)
3    // determine the output device "dev"
4    err = skb_cow_head(skb, LL_RESERVED_SPACE(dev));
```

Listing 2: Simplified code pattern that triggers a double-reallocation of socket buffers in some cases.

needs to add headers but does not need to modify the data. This function does nothing when there is enough headroom. Otherwise, it reallocates more space to accommodate for the required headroom. In that case, the desired headroom size (in bytes) is aligned to the next multiple of a specific value which is totally dependent on the architecture of the CPU and represents its cache line size. In the Linux kernel, the minimum allowed size for a cache line is 32 bytes and can typically be either 64, 128 or even 256 bytes depending on the CPU.

As an example and in order to illustrate, let us assume we want to add 112 bytes to the header of a socket buffer (`skb`), whose current headroom size is 64. Let us also assume a CPU with a 32-byte cache line size. After a call to *skb_cow_head(skb, 112)*, the new headroom size would be 128. The reasoning is the following: (*i*) the CPU cache line size is added to the current headroom size ($64 + 32 = 96$), which is smaller than 112 bytes; (*ii*) the CPU cache line size is re-added to that value ($96 + 32 = 128$), which is now bigger or equal to 112 bytes. Note that after the insertion of the 112 bytes in the header, the new headroom size would be 16 bytes ($128 - 112$).

Double-Reallocation Issue

This paper investigates an issue in the LWT implementation of IOAM, SRv6, and RPL protocols. In normal cases, when an extra header is added to an existing packet, the kernel triggers a reallocation of the socket buffer if its headroom cannot accommodate it. However, in some cases and under specific conditions (see below), the kernel triggers two consecutive reallocations, resulting in performance degradation.

Listing 2 shows the problematic code pattern that is common to the LWT implementation of IOAM, SRv6, and RPL protocols. First, line 1 ensures sufficient headroom in the socket buffer (*skb*) before adding a new header (*hdrlen* bytes), and anticipates the Layer-2 header reconstruction (*mac_len* bytes, *e.g.*, 14 for Ethernet). Then, line 2 inserts the new header in the socket buffer, while line 3 fetches the output device (*dev*) based on the new packet header. This is because the original output device may no longer be the correct one (*e.g.*, an IPv6-IPv6 tunnel encapsulation with different source or destination addresses than the original packet, leading to a different egress interface). Finally, line 4 ensures sufficient headroom in the socket buffer (*skb*) to accommodate the maximum hardware header length of the output device (*dev*), including any extra headroom the NIC may need. The total length returned by the macro *LL_RESERVED_SPACE()* is aligned to 16-byte multiples for machine alignment needs. Indeed, CPUs often take a performance hit when accessing unaligned memory locations. For instance, since an Ethernet header is 14 bytes long, network drivers often end up with the IPv6 header at an unaligned offset. The IPv6 header can be aligned by shifting the start of the packet by 2 bytes.

IOAM	
Inline mode	PTO of 236 or 240 bytes
Encap. mode	PTO of 196 or 200 bytes
SRv6	
Inline mode	None
Encap. mode	For 13, 17, 21, 25, 29, 33, .. segments
Encap. L2 mode	For 13, 17, 21, 25, 29, 33, .. segments
Encap. Red mode	For 14, 18, 22, 26, 30, 34, .. segments
Encap. L2 Red mode	For 14, 18, 22, 26, 30, 34, .. segments
RPL	
Inline mode	None

Table 1: Cases triggering the double-reallocation for an x86 architecture and an Intel XL710 NIC.

To illustrate a case that would trigger a double-reallocation based on Listing 2, let us assume we want to add an extra header (40 bytes) in the socket buffer, whose current headroom size is 22. Let us also assume a CPU with a 32-byte cache line size, and Ethernet as Layer-2. First, in line 1, *hdrlen* is 40 (the extra header size) and *mac_len* is 14 (Ethernet header size). After that, the headroom size becomes 54 (*i.e.*, $22 + 32$, the old headroom size plus the cache line size). Then, in line 2, we insert the extra 40-byte header. Therefore, the headroom size is now 14 (*i.e.*, $54 - 40$, the old headroom size minus the extra header size). Finally, in line 4, we ensure that the headroom can accommodate the hardware header length (*i.e.*, 16 bytes in this case). Since the current headroom size is too small ($14 < 16$), a second reallocation is performed. As a result, the headroom size becomes 46 (i.e., $14 + 32$, the old headroom size plus the cache line size). The root cause of the double-reallocation comes from the difference between line 1 and line 4 in listing 2, where *mac_len* is too generic and often smaller than *LL_RESERVED_SPACE()* due to alignment requirements. A solution to avoid this problem would be to use the latter in both lines, instead of *mac_len*. However, as already mentioned, the output device in line 1 may not be the same after line 3. Therefore, this solution cannot work as is.

As discussed, the double-reallocation mainly depends on the CPU architecture (*i.e.*, L1 cache line size) and, to a lesser extent, on the NIC (*i.e.*, for alignment requirements and headroom allocation). Headroom allocation depends on the NIC. In our case, the network driver (*i40e*) allocates 192 bytes for the headroom[1]. Therefore, before calling *skb_cow_head()*

[1]With *legacy-rx* disabled by default. Otherwise, a cache line sized (*i.e.*, 64-byte for *x86*) headroom is allocated. In both cases,

for the LWT encapsulation, the headroom size of all socket buffers is 206 bytes (*i.e.*, $192 + 14$, because the data pointer was moved beyond the 14-byte Ethernet header). Table 1 summarizes the cases where a double-reallocation happens for the LWT encapsulation of IOAM, SRv6, and RPL protocols. It is based on an *x86* architecture (*i.e.*, 64-byte cache line size), and an `Intel XL710` NIC (*i40e* driver).

Let us take IOAM in Table 1 to illustrate the problem. All four cases produce the same overhead (*i.e.*, *hdrlen* provided to *skb_cow_head()*), that is 256 bytes for the entire Extension Header. Indeed, the Encap mode must include the extra IPv6 header (+40 bytes), which is identical to the Inline mode in terms of overhead. Moreover, a 236-byte and a 240-byte PTO both produce the same overhead, as padding is added to the Extension Header. If we apply the code in Listing 2 to this case, we have: (*i*) an extra 256-byte header to add and the 14-byte Ethernet header to rebuild later, (*ii*) a 206-byte headroom, and (*iii*) a 32-byte cache line size. After line 1, the headroom becomes 270 (*i.e.*, $206 + 64$, the old headroom size plus the cache line size). Then, in line 2, we insert the extra 256-byte header. Therefore, the headroom size is now 14 (*i.e.*, $270 - 256$, the old headroom size minus the extra header size). Finally, in line 4, we ensure that the headroom can accommodate the hardware header length (*i.e.*, 16 bytes in this case). Since the current headroom size is too small ($14 < 16$), a second reallocation is performed. As a result, the headroom size becomes 78 (i.e., $14 + 64$, the old headroom size plus the cache line size).

Because the IOAM PTO is limited to a maximum of 244 bytes, there are no repeated values[2] with a double-reallocation like for SRv6 and RPL (*i.e.* maximum 127 segments, meaning $2,040$ bytes). Nevertheless, the exact same logic applies based on the total overhead for each case, *e.g.*, the total overhead of 21 SRv6 segments (Encap mode) is 384 bytes (*i.e.*, extra IPv6 header plus SRv6 Routing header). Note that the double-reallocation does not happen for SRv6 with the Inline mode because it does not fall on the same boundaries anymore (*i.e.*, -40 bytes compared to the Encap mode with the extra IPv6 header). On the other hand, RPL (only the Inline mode is implemented in the kernel) comes with a compression mechanism, which makes it theoretically possible to trigger the double-reallocation issue. However, because `iproute2` [11] has an input buffer limited to a maximum of $1,024$ characters, we could not add enough segments to make the total overhead large enough to trigger the issue.

Overall, it is possible to generalize the problem to other CPU architectures: the smaller the cache line size, the more often the double-reallocation of socket buffers will occur. For instance, having a 32-byte cache line would trigger the issue twice as frequently compared to a 64-byte cache line. E.g., with a 32-byte cache line, the double-reallocation would happen with SRv6 (Encap mode) for the following number of segments: 11, 13, 15, 17, 19, 21, 23, 25, etc (instead of 13, 17, 21, 25, etc with a 64-byte cache line as in Table 1).

the double-reallocation happens because 192 is a multiple of 64.

[2]There would be more with *legacy-rx* enabled, depending on the cache line size.

```
1    err = skb_cow_head(skb, hdrlen + dst_dev_overhead(cache
         , skb));
2    // add the new header ("hdrlen" bytes)
3    // determine the output device "dev" if cache empty
4    err = skb_cow_head(skb, LL_RESERVED_SPACE(dev));
```

Listing 3: Simplified code pattern to mitigate the double-reallocation of socket buffers.

```
static inline unsigned int dst_dev_overhead(
    struct dst_entry *dst, struct sk_buff *skb)
{
  if (likely(dst))
    return LL_RESERVED_SPACE(dst->dev);

  return skb->mac_len;
}
```

Listing 4: Code of the new dst_dev_overhead() function.

Mitigation Solution

The solution proposed in this paper leverages the existing cache system in the LWT implementation of IOAM, SRv6, and RPL protocols. As a reminder, a LWT encapsulation is attached to a route. Therefore, before a packet matches such a route, the cache which is supposed to contain the corresponding output device is empty. Once a first packet matches the route, the LWT encapsulation is applied to the packet, and the output device is fetched and stored in the cache. After that, the cache is directly used every time another packet matches the same route, without fetching the output device after the LWT encapsulation anymore.

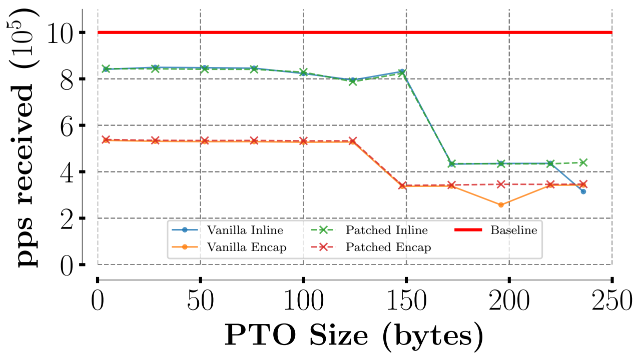
Listing 3 shows the simplified code pattern found in listing 2, with the mitigation solution applied. Line 1 uses a new function named *dst_dev_overhead()*. The code of this new function is shown in listing 4. Its purpose is to return the headroom size required by the output device (for the Layer-2 header and any extra data needed by the NIC) when the cache is not empty. Otherwise, if the cache is empty, the function returns the generic *mac_len*. Therefore, the cache is now checked before the LWT encapsulation. The logic behind lines 2, 3 and 4 is the same as previously.

As a result, the first packet that matches a route with an attached LWT encapsulation would see an empty cache. Should all specific conditions be met for the double-reallocation to happen, the issue would persist for that first packet only. Indeed, it is impossible to avoid it in that case because the cache is needed, which is not possible for the very first packet. On the other hand, subsequent packets are not impacted by the double-reallocation anymore, thanks to the proposed mitigation solution that leverages the cache.

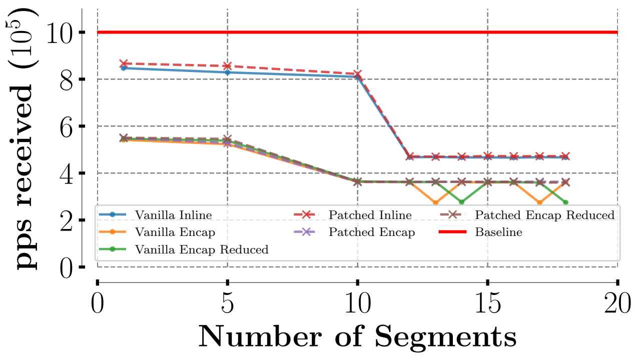
Performance Results

Methodology

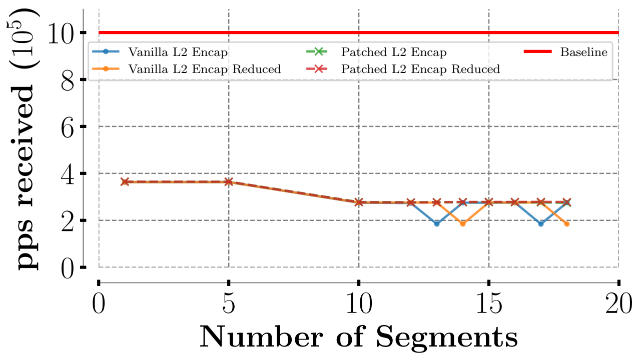
Performance evaluation follows the same methodology across all experiments. We use Trex [5], a high-performance traffic generator, to transmit at line rate hand-crafted packets



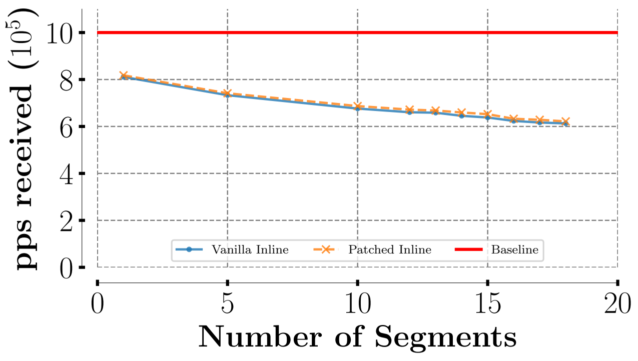
(a) IOAM



(b) SRv6



(c) SRv6 (L2 mode)



(d) RPL

Figure 6: Performance evaluation.

to the Device Under Test (DUT)[3], which is a Linux machine with a recent kernel[4] configured to forward traffic back to the generator in a port-to-port setup.

The key performance metric is the maximum number of packets per second (pps) the DUT can forward. We compare performance of a vanilla Linux kernel with our patched version. The DUT baseline (IPV6 packet forwarding rate) on a vanilla kernel is 1,000,000 packets per second, as indicated by the red lines in Fig. 6. For each value on the plots, 5 experiments are performed. Confidence intervals around the mean are computed but too tight to appear in the plots.

Results

Fig. 6a shows the forwarding capabilities for increasing sizes of the IOAM PTO. Performance degradation due to the double-reallocation issue is clearly visible with a vanilla kernel: one can see a forwarding rate drop occurring for PTOs of 196 bytes with the Encap mode, and for PTOs of 236 bytes with the Inline mode. On average, performance decreases by 27.1% with a vanilla kernel when the issue occurs. The issue is completely mitigated in the patched version.

[3]DUT specifications: *x86 Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz*, *16 GB* RAM, and *Intel XL710 Dual Port 40G QSFP+* NIC (i40e driver)

[4]Linux kernel version 6.12.

Fig. 6b and 6c show the forwarding capabilities for increasing number of segments in the SRv6 header. In particular, Fig. 6c specifically shows performance of SRv6 in L2 mode. In both figures, the vanilla kernel exhibits noticeable performance drops at specific segment counts. These drops occur at segment counts of 13 and 17 for the Encap mode (both normal and L2 versions), and at segment counts of 14 and 18 for the reduced ("Red") Encap mode (both normal and L2 versions). On average, the vanilla kernel experiences a 28.8% reduction in performance when the issue occurs. The issue is completely mitigated in the patched version.

Fig. 6d demonstrates that performance remains consistent across the vanilla and patched kernels for RPL. This is because RPL is unaffected by the double-reallocation issue, as explained previously.

As a summary, one can conclude that the patch proposed in this paper effectively mitigates the double-reallocation issue, without affecting performance for normal cases where the double-reallocation does not happen.

Conclusion

In this paper, we investigated a performance issue in the Linux kernel LWT implementation of IOAM, SRv6, and RPL protocols. The issue, which occurs under specific conditions, causes a double-reallocation of socket buffers instead of a single one, resulting in significant performance degradation of up to 28.8%.

We proposed a patch to mitigate the double-reallocation issue and, through performance comparisons between a vanilla kernel and the patched version, we demonstrated that the observed performance degradation was effectively mitigated in IOAM and SRv6. Although RPL was not directly affected by the issue, its implementation shared similar code patterns. As a preventive measure, we applied the patch to RPL as well, ensuring that potential future problems are avoided.

Source Code

The patch proposed in this paper has been merged in the Linux kernel. A link to the corresponding thread on the mailing list archive will be provided when the paper is accepted.

Acknowledgments

This work has been supported by the CyberExcellence project, funded by the Walloon Region, under number 2110186, and the Feder CyberGalaxia project.

References

[1] Alexander, R.; Brandt, A.; Vasseur, J.; Hui, J.; Poster, K.; Thubert, P.; Levis, P.; Struik, R.; Kelsey, R.; and Winter, T. 2012. RPL: IPv6 routing protocol for low-power and lossy networks. RFC 6550, Internet Engineering Task Force.

[2] Bhandari, S., and Brockners, F. 2023. IPv6 Options for In Situ Operations, Administration, and Maintenance (IOAM). RFC 9486, Internet Engineering Task Force.

[3] Brockners, F., and Bhandari, S. 2023. Network Service Header (NSH) Encapsulation for In Situ OAM (IOAM) Data. RFC 9452, Internet Engineering Task Force.

[4] Brockners, F.; Bhandari, S.; and Mizrahi, T. 2022. Data Fields for In Situ Operations, Administration, and Maintenance (IOAM). RFC 9197, Internet Engineering Task Force.

[5] Cisco. 2015. Trex. [Last Accessed: April 29th, 2024].

[6] Deering, S., and Hinden, B. 2017. Internet protocol, version 6 (IPv6) specification. RFC 8200, Internet Engineering Task Force.

[7] Farrel, A., and Bonica, R. 2017. Segment routing: Cutting through the hype and finding the IETF's innovative nugget of gold. *IETF Journal* 13(1).

[8] Filsfils, C.; Previdi, S.; Grinsberg, L.; Decraene, B.; Likowski, S.; and Shakir, R. 2018. Segment routing architecture. RFC 8402, Internet Engineering Task Force.

[9] Filsfils, C.; Dukes, D.; Previdi, S.; Leddy, J.; Matsushima, S.; and Voyer, D. 2020. Ipv6 segment routing header (srh). RFC 8754, Internet Engineering Task Force.

[10] Filsfils, C.; Camarillo, P.; Leddy, J.; Voyer, D.; Matsushima, S.; and Li, Z. 2021. Segment routing over IPv6 (SRv6) network programming. RFC 8986, Internet Engineering Task Force.

[11] Hemminger, S., and contributors. 2024. iproute2. `https://wiki.linuxfoundation.org/networking/iproute2`. Linux networking utilities.

[12] Herbert, T., and Lapukhov, P. 2018. Identifier-locator addressing for IPv6. Internet Draft (Work in Progress) draft-herbert-intarea-ila-01, Internet Engineering Task Force.

[13] Hui, J.; Vasseur, J.; Culler, D.; and Manral, V. 2012. An IPv6 Routing Header for Source Routes with the Routing Protocol for Low-Power and Lossy Networks (RPL). RFC 6554, Internet Engineering Task Force.

[14] kernel development community, T. 2024. struct sk_buff. `https://docs.kernel.org/networking/skbuff.html`.

[15] Klassert, S., et al. 2018. Virtual xfrm interfaces.

[16] Muthukrishnan, K., and Malis, A. 2000. A core MPLS IP VPN architecture. RFC 2917, Internet Engineering Task Force.

[17] Roopa, R., and G., T. 2015. Lightweight & flow based tunneling.

[18] Song, H.; Gafni, B.; Brockners, F.; Bhandari, S.; and Mizrahi, T. 2022. In Situ Operations, Administration, and Maintenance (IOAM) Direct Exporting. RFC 9326, Internet Engineering Task Force.

[19] Ventre, P. L.; Salsano, S.; Polverini, M.; Cianfrani, A.; Abdelsalam, A.; Filsfils, C.; Camarillo, P.; and Clad, F. 2020. Segment routing: A comprehensive survey of research activities, standardization efforts, and implementation results. *IEEE Communications Surveys & Tutorials* 23(1):182–221.

[20] Viswanathan, A.; Rosen, E. C.; and Callon, R. 2001. Multiprotocol label switching architecture. RFC 3031, Internet Engineering Task Force.