Implementation of IPv6 IOAM in Linux Kernel

Justin Iurman[*], **Benoit Donnet**[*], **Frank Brockners**[‡]
[*] Université de Liège, Montefiore Institute – Belgium
[‡] Cisco

Abstract

In-situ Operations, Administration, and Maintenance (IOAM) is currently under standardization at the IETF. It allows for collecting telemetry and operational information along a path, within packets, as part of an existing (possibly additional) header. This paper discusses the very first implementation of IOAM for the Linux kernel with IPv6 as encapsulation protocol. We also evaluate our implementation, available as open source, under a controlled environment.

Introduction

Operations, Administration, and Maintenance (OAM) refers to a set of techniques and mechanisms for performing fault detection and isolation and for performance measurements. Throughout the years, multiple OAM tools have been developed for various layers in the protocol stack [13], going from basic `traceroute` [16] to *Bidirectional Forwarding Detection* (BFD [10]). Recently, OAM has been pushed further through *In-Situ* OAM (IOAM) [2, 4, 3]. The term "In-Situ" directly refers to the fact that the OAM and telemetry data is carried within packets rather than being sent through packets specifically dedicated to OAM. The IOAM traffic is embedded in data traffic, but not part of the packet payload.

The well-known IPv4 *Record-Route option* [15] can be considered as an IOAM mechanism. However, compared to that, IOAM comes with multiple advantages: (i) IOAM is not limited to 40 BYTES as for the Record-Route option (recording so a maximum of 9 IPv4 addresses); (ii) IOAM allows different types of information to be captured, including not only path tracing information but additional operational and telemetry data such as timestamps, sequence numbers, or even generic data such as queue size or geo-location of the node that forwarded the packet; (iii) IOAM allows one to record the path taken by a packet within a fixed amount of added data; (iv) IOAM data fields are defined in a generic way so that they are independent from the protocol that carries them; (v) finally, IOAM offers the ability to actively process information in the packet. For instance, IOAM allows one to prove in a cryptographically secure way that a packet really followed a pre-defined path using a traffic steering method, such as service chaining (e.g., *NFV service chaining*) or traffic engineering (e.g., through *Segment Routing* [9]).

In a nutshell, IOAM gathers telemetry and operational information along a path, within packets, as part of an exist-

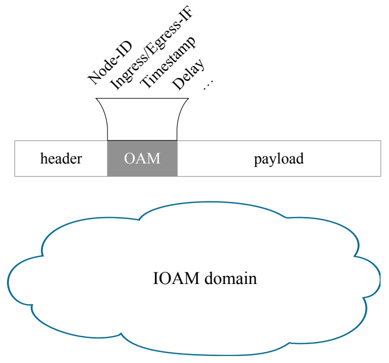


Figure 1: High-level view of IOAM

ing (possibly additional) header, as shown in Fig. 1. It is encapsulated in IPv6 packets as an *IPv6 HopByHop extension header* [6, 1]. Typically, IOAM is deployed in a given domain, between the INGRESS and the EGRESS or between selected devices within the domain. Each node involved in IOAM may insert, remove, or update the extension header. IOAM data is added to a packet upon entering the domain and is removed from the packet when exiting the domain. There exist four IOAM types for which different IOAM data fields are defined. (i) the *Pre-allocated Trace Option*, where space for IOAM data is pre-allocated; (ii) the *Incremental Trace Option*, where nothing is pre-allocated and each node adds IOAM data while expanding the packet as well; (iii) the *Proof of Transit* (POT) and, (iv) the *Edge-to-Edge* (E2E) Option. Trace and POT options are both embedded in a *HopByHop extension header*, meaning they are processed by every node on the path, while E2E option is embedded in a *Destination extension header*, which means it is only processed by the destination node.

IOAM data fields are defined within IOAM *namespaces*, that are identified by a 16-bit identifier. They allow devices that are IOAM capable to determine, for example, whether IOAM option header(s) need to be processed, which IOAM option headers need to be processed/updated in case of multiple IOAM option headers, or whether IOAM option header(s) should be removed. IOAM namespaces can be used by an operator to distinguish different operational domains. Devices at domain boundaries can filter on namespaces to provide for proper IOAM domain isolation. They also provide additional

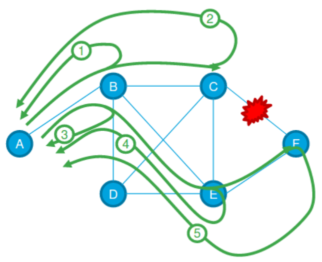


Figure 2: Failure Detection with IOAM.

context for IOAM data fields, ensuring IOAM data is unique, as well as allowing to identify different sets of devices.

Traditionally to detect and isolate network faults, `ping` and `traceroute`, or even bidirectional forwarding detection (BFD) [10], are used. But in a complex network with large number of U/E-CMP being available, it would be difficult to detect and isolate them. Also detecting loss/reordering/duplication of packets becomes much harder. Currently, failure detection takes ten of seconds in large networks, while failure isolation (using `ping` and `traceroute`) takes several minutes [8, 7]. Indeed, probing based on `traceroute` is slow. Therefore, the objective is to significantly improve failure detection and isolation through efficient network probing, specially for hyper giant distribution networks (HGDNs, such as Facebook, Google, or Netflix) and large data center networks (DCNs).

This is exactly a context in which an active network probing relying upon UDP probes with IOAM can find a suitable usage. If one encounters connectivity issues between individual nodes, then IOAM tracing could be enabled between those nodes to understand where things are going wrong. This is of particular benefit, specially when HGDNs and DCNs make use of equal-cost multipath (ECMP).

With IOAM, one can identify the exact path, something which is really hard to do with standard `traceroute`. By using the IOAM loopback flag, an issue can be identified within a single packet RTT, as illustrated in Fig. 2. Additional use cases for IOAM are m-anycast service (intelligent micro-service selection and load-balancing) [5], service and quality assurance (proving the traffic follows a pre-defined path through POT), or high precision congestion control [12].

In this paper, we provide the first implementation of IPV6 IOAM inside the Linux kernel. This implementation is important as it provides to operators and IETF an insight into IOAM practical aspects. In addition, IOAM can complement *Layer5-7* tracing solutions such as OpenTelemetry [14] to create a comprehensive *Layer2-7* tracing solution.

This paper describes our implementation[1] inside the Linux kernel version 4.12. We designed our implementation to be as efficient as possible, through a new kernel module. This paper also discusses early performance results of our IOAM implementation. In particular, we show that inserting IOAM headers and data may reduce the bandwidth capabilities of an IOAM domain but could also stay efficient thanks to some measured compromises. The additional delay introduced by IOAM processing is therefore quite reasonable.

[1]Our implementation is available as open source under the terms of the GNU General Public License version 2 (GPL-2.0). See `https://github.com/IurmanJ/kernel_ipv6_ioam`

```
struct ioam_node {
  __u32 ioam_node_id; //IOAM node identifier
  unsigned int if_nb; //number of IOAM interfaces
  unsigned int ns_nb; //number of known IOAM namespaces
  unsigned int encap_nb; //number of IOAM insertions
  unsigned int encap_freq; //frequency of IOAM insertions
  struct ioam_interface ifs[IOAM_MAX_IF_NB];
  struct ioam_namespace nss[IOAM_MAX_NS_NB];
  struct ioam_encapsulate encaps[IOAM_MAX_NS_NB];
};
```

Figure 3: IOAM node registration structure.

Linux-Kernel Implementation

In this section, we carefully describe how we have implemented IOAM within the Linux kernel 4.12.[1] We first cover the IOAM node registration step as well as any IOAM resources allocation. We next explain how packet parsing can be done efficiently for `IPv6 Extension Headers`. We finally focus on how IOAM headers are inserted and removed from packets. Our implementation is based on IOAM drafts [1, 3], respectively versions `02` and `06`.

User Space API

The registration process is required for a node to enable IOAM. As long as it is not the case, the node will drop anything that is IOAM related. For that purpose, a new `ioctl` has been created. In order to fit in the kernel code, we provide a `uapi` (*user space API*) to facilitate the `ioctl` call. This also eases the registration process from a user point of view. Indeed, as a good practice, everything inside the `uapi` has been documented, since it is what users can see and include in their programs. **Important note**: in the subsequent versions, the `ioctl` call will be replaced and we will use `genetlink` and `rtnetlink` instead, as we do believe that it would ease the acceptance of the code inside the kernel as well as being more appropriate. Fig. 3 shows the main structure of the IOAM `uapi`.

The list of IOAM interfaces (`ifs`) contains a mapping between a device and its IOAM identifier chosen by the operator, as well as its IOAM role $\in \{none, ingress, egress\}$. A `none` role means that the interface is an IOAM domain boundary not allowing incoming IOAM traffic (default role), while `ingress` and `egress` respectively mean that the interface allows for incoming and outgoing IOAM traffic. An IOAM interface can play the role of both `ingress` and `egress` at the same time. If an IOAM interface owns the `egress` role, the operator must also specify an `IPv6` address being the tunnel destination. Indeed, as `RFC 8200`[6] does not allow for in-flight insertion/removal of extension headers, we must create a tunnel (IPV6-IN-IPV6) when the traffic is coming from outside in order to insert IOAM headers. The list of IOAM namespaces (`nss`) represents all namespaces known by the node and contains per-namespace data such as its identifier, as well as whether it should be removed by the node or not. A

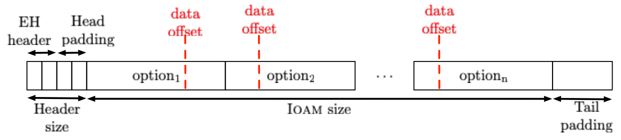


Figure 4: IOAM buffer representing an IPv6 Extension Header.

node will ignore IOAM data from an unknown IOAM namespace. Be careful not to mix the notion of IOAM namespaces and Linux namespaces, as they are totally different and unrelated. Note that this implementation is Linux namespaces compliant. The list of IOAM insertions (`encaps`) contains IOAM data that should be inserted in packets, along with their IOAM namespace identifier and the egress IOAM interface. The same namespace identifier can be inserted multiple times, for different IOAM options, as well as several IOAM options in the same namespace.

This structure provides a complete configuration of an IOAM node and allows for strict filtering due to IOAM interface roles and IOAM egress interfaces when inserting IOAM headers. The operator has all the possibilities, such as configuring overlapping IOAM namespaces, unidirectional, or bidirectional IOAM flows.

Node Registration

Calling the `ioctl` starts several steps. First, the IOAM kernel module parses and validates data. Then, it builds internal structures based on received data and every single IOAM resource is allocated per kernel network namespace. The main goal is to design a solution to allocate any resources needed by IOAM, at registration time, in order to keep packet processing as fast as possible.

The very first important resource to store is the list of known IOAM namespaces, at node level, as all interfaces must be able to access it. Since the IOAM namespace lookup occurs each time a packet with IOAM data is processed, a kernel hashtable, with the IOAM namespace identifier as the key, has been used for that purpose. The trade-off is to find a balance between potential collisions and memory allocation. We achieve this by forcing the hashtable size being four times the maximum number of allowed IOAM namespaces. In the near future, we will investigate the pros and cons of replacing fixed-size by dynamic size hashtables. The IOAM node identifier is also stored along with the hashtable, as well as pre-allocated paddings to speed up IOAM options alignment (see Sec. “Header Insertion”).

The other important resource to store is an optimized buffer that represents an `Extension Header` with IOAM (either a `Hop-by-Hop` or `Destination` option) to be inserted, built from registration data. Fig. 4 illustrates how the buffer is structured and how it works (more details on this buffer in Sec. “Header Insertion”). It is allocated twice per IOAM encap interface, one for a `Hop-by-Hop` option and another for a `Destination` option. The IOAM identifier, role, counter for frequency insertions, and the tunnel destination are also stored per interface.

```
struct ioam_parsed_eh {
  u16 size; //extension header size
  u8 pad_size; //total EH padding size
  u16 decap_size; //total size of IOAM data to remove
  u8 free_idx;
  struct ioam_block decaps[IOAM_MAX_NS_NB];
};
```

Figure 5: Structure with additional data for an Extension Header parsing.

EH Parsing

`IPv6 Extension Headers` already come with parsing mechanisms inside the kernel based on RFC8200 [6]. However, for speeding up as much as possible the IOAM header decapsulation process, we need to store additional information. This is the objective pursued by the `ioam_parsed_eh` structure (see Fig. 5). It helps to easily remove IOAM from packets, while remaining efficient. The usefulness of each field will be detailed in Sec. “Header Removal”.

The field `decaps` represents the list of $\{offset, size\}$ IOAM blocks to be removed, with `free_idx` giving the very next free slot in the array, as well as the number of IOAM namespaces to be removed. Having a fixed-size array is a choice we made to avoid allocation during critical processing. Note that defined boundaries in the IOAM user space API are realistic and as such are not a problematic limit.

As the IOAM header decapsulation process needs this data, we must store the structure while keeping in mind the main goal of avoiding any allocation. Initially, we were storing the structure per network devices (`struct net_device`), where it was pre-allocated at the registration. However, we found out that packet queues were multi-threaded in the kernel, which prevented this solution from working. Ideally, the structure should follow its specific packet, just like if it was attached to it, but that would also mean an allocation during packet processing. We started to look inside the `sk_buff` structure that represents a network packet, and discovered that a private user data space was available. This field is called the `control buffer` (`cb`) and is said to be free to be used by every layer, where one can put private variables there. But the current layer, IPv6, already uses this field for storing an internal structure (`inet6_skb_parm`). Finally, in order to integrate IOAM as efficiently as possible within the kernel, we decided to store a pointer to `ioam_parsed_eh` inside `cb`. It means less memory to allocate per packet (four or eight octets, depending on the architecture), which is critical, especially when this additional field aims at being used “locally”. A concrete and simplified example of how it works is shown in Fig. 6. Thanks to this technique, we first respect our objective to not allocate anything during packet processing and, secondly, multi-threaded packet queues are now supported.

Header Insertion

The IOAM header insertion process happens at the output, which makes sense from a semantic point of view. Here, two different scenarios are possible. Note that, in both cases,

```
struct ioam_parsed_eh parsed_eh;
//Init parsed_eh fields before parsing
IP6CB(skb)->ioam = &parsed_eh;

if (ip6_parse_tlv(skb)){
  //decap, if any, based on parsed_eh
}
```

Figure 6: Simplified example of IPv6 input for a Hop-by-Hop.

IOAM will not be added in a packet if the new size is bigger than the MTU. **Important note**: we plan to use the lightweight tunnels API available in the kernel to handle this more cleanly.

The first scenario applies to traffic generated by the node itself. In that case, IOAM headers are directly inserted. Note that it is only true when there is no Hop-by-Hop or Destination option already present in a packet. The obvious reason for such a restriction is to not play and slow down sensitive packets such as Router Alert. Therefore, we simply insert the entire buffer shown in Fig. 4, right after the IPv6 header. The first two octets represent the EH header, while the two next octets are padding. Indeed, IOAM options require a $4n$ alignment [1, 3], which is specified using the notation $xn + y$, meaning the Option Type must appear at an integer multiple of x octets from the start of the EH header, plus y octets. Following that, each IOAM option is inserted, as well as a tail padding such that the complete EH length is an integer multiple of 8 octets [6].

The second scenario applies to in-transit traffic. In that case, IOAM headers are encapsulated (IPv6-in-IPv6 tunnel) to make sure it is compliant with RFC 8200. Therefore, we again insert the entire buffer shown in Fig. 4, this time *before* the IPv6 header which becomes an inner header. A similar IPv6 header is added in front of the buffer so that the encapsulation is complete. The tunnel destination is another node inside the IOAM domain. Note that it is hard to find a suitable solution for overlapping and/or nested IOAM namespaces (aka tunnels). Indeed, we recommend operators to only use a single encapsulation from domain border to domain border, if possible, to avoid inner IOAM headers being ignored and/or leaked out.

The strength of this algorithm is that it covers both scenarios, which makes the specific host-to-host use case possible. Also, it does not require any allocation, as buffers are already allocated and filled at registration time. However, an implicit allocation could occur if the required extra space is bigger than the free space in the packet buffer. A solution would be to increase the needed_headroom field of each IOAM encap interface (struct net_device) to allocate a bit more for each packet and as such avoid this situation. Again, this is a trade-off between memory and performance and the choice is up to the operator.

Data Insertion

The insertion of IOAM data is only applied by a node if the input interface has an IOAM ingress role *and* if the IOAM namespace is known. The latter requires a lookup inside the hashtable containing all known IOAM namespaces by the node. If those conditions are respected, the node inserts its data, depending on what data was required.

Note that IOAM data insertion not only happens when receiving a packet with IOAM but also after IOAM header insertion (see Sec. "Header Insertion"). Indeed, for the latter, a node that adds IOAM headers inside a packet (either by inserting or encapsulating them), must insert its IOAM data too.

Header Removal

The IOAM header removal process is only applied on Hop-by-Hop options when the node is not the destination. Indeed, removing IOAM headers when the node is the destination (Hop-by-Hop or Destination option) would cause an additional and unnecessary workload. This process becomes straightforward thanks to data provided by the improved parsing. We know the total size of IOAM data blocks to be removed as well as their respective positions and sizes, as shown in Fig. 5. One can distinguish two different cases.

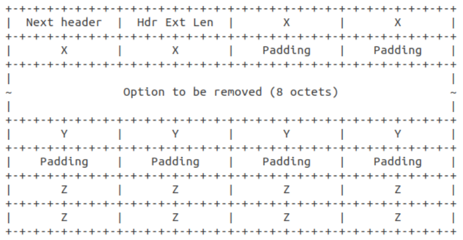
The first and fastest case is when the entire Extension Header should be removed. It is possible to quickly check that since we know the size of data to be removed, as well as the total padding size and the Extension Header size. If $\texttt{decap\_size} + \texttt{pad\_size} == \texttt{EH.size} - 2$, then the entire Extension Header should be removed (minus 2 to not take the header into account). Right now, this is how it is implemented. However, still trying to be compliant with RFC 8200, we wonder if it would not be a better choice to have an "empty" Extension Header instead of removing it altogether, though this could sound crazy.

For the other case, it will omit IOAM blocks that must be removed while shifting and copying data back, and append tail padding if needed to keep alignment.

To remain effective, options are not reordered. As such, the arrangement may not be optimal. However, we can't assume that options after the removed one, if any, are still aligned. Therefore, we need to merge possible padding before and after the removed option into an accurate padding, so that the alignment is respected. But this is not enough because next options, if any, could still be misaligned. Let's assume the following alignment requirements 2n, 4n, and 8n respectively for options X, Y, and Z. The example shown in Fig. 7 would work, which is not true for the other shown in Fig. 8. Indeed, after the removal, the option Z will no longer be aligned on an 8-octet boundary. In this case, knowing the alignment of the very next option after the removed one wouldn't help. We would also need to know the alignment of any next options and try to realign everything option by option, which is not acceptable from a performance point of view. Instead, a good compromise that works for all cases is to automatically assume an 8-octet boundary. As a result, unnecessary padding could be introduced (4 octets at most), which is acceptable.

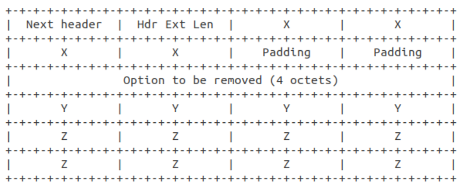
Evaluation

In this section, we make a first attempt at evaluating IOAM performances inside the Linux kernel. We first present our testbed and, then, discuss our results.



```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Next header  |  Hdr Ext Len  |       X       |       X       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       X       |       X       |    Padding    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                 Option to be removed (8 octets)               ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Y       |       Y       |       Y       |       Y       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Padding    |    Padding    |    Padding    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Z       |       Z       |       Z       |       Z       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Z       |       Z       |       Z       |       Z       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 7: Option removal, valid scenario for alignment.



```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Next header  |  Hdr Ext Len  |       X       |       X       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       X       |       X       |    Padding    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 Option to be removed (4 octets)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Y       |       Y       |       Y       |       Y       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Z       |       Z       |       Z       |       Z       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Z       |       Z       |       Z       |       Z       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 8: Option removal, invalid scenario for alignment

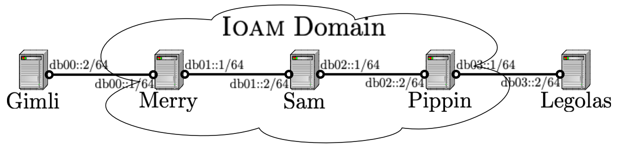


Figure 9: Topology used for measurements.

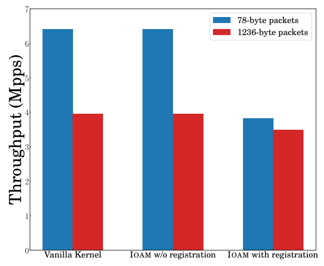


Figure 10: Baseline throughput comparison.

Testbed and Measurement Methodology

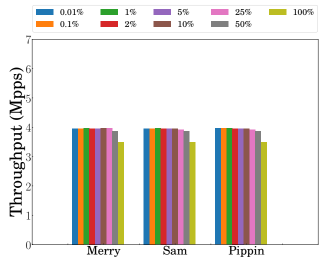
Fig. 9 shows the testbed we built for IOAM performance evaluation. It is made of five machines, three of them playing the role of the IOAM domain. The remaining two are respectively positioned on each side of the IOAM domain to send and receive traffic. Each machine, configured to maximize its performance (e.g., maximum CPU frequency), is directly connected to its peers and is equipped with Intel XL710 2x40GB QSFP+ NICs having Receive-Side Scaling (RSS) enabled and firmware up-to-date, as well as the latest i40e driver. `Gimli`, the traffic generator, is based on Intel Xeon CPU E5-2683 v4 at 2.10GHz, 16 Cores, 32 Threads, 64GB RAM. `Merry` and `Pippin`, the two IOAM domain boundaries, are both based on Intel Xeon CPU E5-2630 v3 at 2.40GHz, 8 Cores, 16 Threads, 64GB RAM. Their purpose are respectively to encapsulate and decapsulate IOAM headers as well as inserting their IOAM data. `Sam` is based on Intel Xeon CPU E5-2630 v3 at 2.40GHz, 8 Cores, 16 Threads, 32GB RAM. Its sole purpose is to insert its IOAM data. As for `Legolas`, the receiver, it is based on Intel Xeon CPU E5-2620 v4 at 2.10GHz, 16 Cores, 32 Threads, 128GB RAM.

In order to send traffic at line rate (40 Gbps), we use `pktgen` [11] which is an alternative tool included in the kernel that bypasses the Linux kernel network stack by generating packets directly on top of the device driver in kernel space. Indeed, reaching line rate with userspace tools like `iperf3` can be painful. For instance, with the latter, we reached ~37 Gbps, at best, while we managed to *constantly* send more than 39 Gbps with `pktgen`. Even if `pktgen` is only able to generate UDP packets, we are fine with it as we want to measure the additional processing overhead introduced by IOAM. We also use `dstat` [17] on each machine to log RX and TX every seconds.

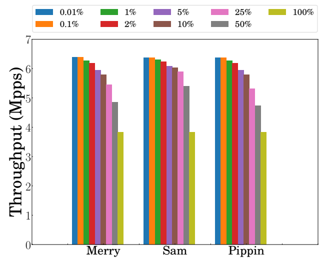
Each experiment lasts 30 seconds and is run 20 times. We determine 95% confidence intervals for the mean based on the Student t distribution. However, as they are too tight to appear on the plots, we took them off. For each experiment, there are two sub-experiments, one for the traffic involving 78-byte packets and another for 1236-byte packets. The reason for this is to measure the IOAM processing overhead for both MTU-sized and very small packets, with the latter being useful to show some additional load brought by IOAM that the former may not reveal. For MTU-sized packets, we chose 1,236 bytes because the maximum IOAM overhead for all of our tests is 264 bytes, which gives 1,500 bytes (default MTU). Indeed, having a common base allows us to equally compare each experiment and measure the same thing all along. For small packets, we use 78 bytes as this is the shortest that `pktgen` can send: 14 bytes for Ethernet, 40 bytes for IPv6, 8 bytes for UDP, and 16 bytes for `pktgen`. Each plot in the next subsection provides results in Mpps (Million Packets Per Second).

Results

We first examine the throughput baseline on a vanilla kernel 4.12, and compare it to the same kernel patched with IOAM. For the latter, we distinguish two cases. Firstly, IOAM without registration, meaning that IOAM was compiled and included in the kernel but disabled. The goal is to measure the impact of the patch being totally passive. Secondly IOAM with registration, meaning that not only IOAM was compiled and included in the kernel but is also enabled. For the latter, we consider the base case with the encapsulation of a single IOAM namespace containing a single IOAM trace option, leading to an overhead of 72 bytes per packet in total. Fig. 10 shows the results for 78-byte (blue bar) and 1236-byte (red bar) packets. For both cases, there is no decrease between a vanilla kernel and the patched IOAM kernel with IOAM disabled. We see that line rate is sustained with 1236-byte packets while ~6,400,000 pps is the upper limit the kernel can handle for 78-byte packets. Indeed, the smaller the packets the more processing for the kernel. Unsurprisingly, the loss comes with IOAM enabled. For 1236-byte packets, we lose ~400.000 pps (~4 Gbps), which represents a loss of 10%. As for 78-byte packets, we lose ~2.600.000 pps (~1.5 Gbps), which represents a loss of 40%. The latter gives more packet loss but involves smaller packets. However, the "IOAM enabled" experiment is quite unrealistic. This is explained in



(a) 1236-byte packets.



(b) 78-byte packets

Figure 11: Frequency of IOAM insertion.

```
ioam_pkt_size = pkt_size + ioam_overhead

pps = (bytes * (1 / frequency) / ioam_pkt_size)
+ (((bytes * (frequency - 1)) / frequency) / pkt_size)
```

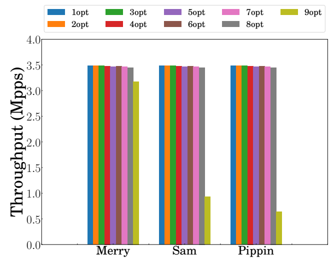
Figure 12: PPS estimation based on the IOAM frequency of insertion

the next experiment.

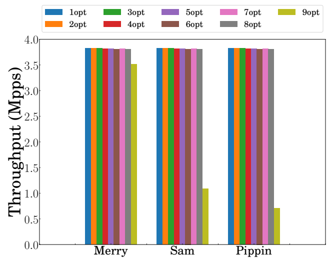
IOAM aims at inserting telemetry data within IPV6 packets. We now investigate the impact of this insertion process, as a fraction of IPV6 packets. In particular, we insert IOAM data in IPV6 packets for 0.01%, 0.1%, 1%, 2%, 5%, 10%, 25%, 50% and 100% (i.e., IOAM data is injected in every packet) of traffic traversing the IOAM domain. Here, we consider again the base case with the encapsulation of a single IOAM namespace containing a single IOAM trace option, which gives an overhead of 72 bytes per packet in total. Packets per second are estimated, from bytes per second obtained during measurements, with the formula provided in Fig. 12. This is based on the frequency of IOAM insertion in order to respect the proportion of the traffic, making a difference between packets with or without IOAM inside.

Fig. 11a shows for 1236-byte packets that line rate is sustained until 10% of insertions. It then slowly starts to drop for higher frequency of insertions. Overall, one can see that only a full insertion (100%) gives a noticeable difference. From 0.01% (which is almost the same as "no insertion") to 100% of insertions, we logically observe the same loss as for previous experiment which is one-tenth of the traffic. The equivalent result for 78-byte packets is illustrated in Fig. 11b and shows also the same loss of 40% as previously. As stated in the previous experiment, the "100% IOAM insertion" case is unrealistic. From those results, we expect operators to not insert telemetry data in every packet traversing their domain as the data processing load would quickly become unmanageable. We rather expect IOAM to be applied on a tiny portion of the traffic or for very specific and limited use cases, which is the purpose of IOAM, corresponding so to no more than 10% or 25%. In such a case, the impact on the IOAM domain throughput is quite limited. On top of that, one could also imagine having a dynamic threshold to insert IOAM depending on the current throughput and packet sizes.

Now that we have an idea of the impact of IOAM data insertion, let us have a look on the impact of increasing the number of IOAM options and the number of IOAM namespaces. The next two experiments not only compare the impact of IOAM overhead but also some additional processing depending on the case, which is explained below. All those experiments allow us to determine a good compromise between the frequency of insertion, the number of options, and the number of namespaces, to keep this process as efficient as possible.

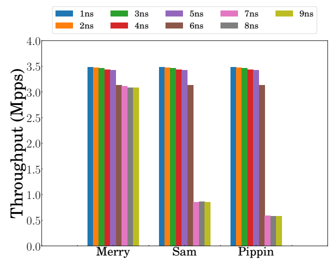


(a) 1236-byte packets.

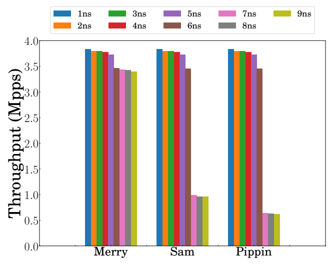


(b) 78-byte packets

Figure 13: Number of IOAM trace options.



(a) 1236-byte packets.



(b) 78-byte packets

Figure 14: Number of IOAM namespaces.

Fig. 13 shows the results of increasing the number of options, respectively for 1236-byte (Fig. 13a) and 78-byte (Fig. 13b) packets. We insert 1 $\rightarrow$ 9 options in every packets, inside a single IOAM namespace, which corresponds to the extreme (and worst) case discussed previously. Those options can have different sizes and give the following overhead: 72 bytes, 80 bytes, 96 bytes, 104 bytes, 120 bytes, 128 bytes, 152 bytes, 176 bytes, and 200 bytes. This includes the outer IPV6 header as well as the IOAM overhead (Hop-by-hop containing 1 $\rightarrow$ 9 options for three hops, and padding). The same behavior is observed on both figures, which is stable until the ninth option on `Merry`. We do believe that this is due to the fact that the socket buffer has no free room enough during encapsulation. Therefore, it requires a re-allocation of extra space by the kernel to be able to deal with the entire IOAM data. This is something we can not avoid. However, we could give the operator the possibility to increase `skb->needed_headroom` to reduce the re-allocation. Be aware that this is again a trade-off between memory and performance. Note that we do not have a strong explanation about the drop on `Sam` for the ninth option.

Fig. 14 shows the results of increasing the number of namespaces, respectively for 1236-byte (Fig. 14a) and 78-byte (Fig. 14b) packets. We insert 1 $\rightarrow$ 9 namespaces in every packets, each containing a single IOAM option, which gives the following overhead: 72 bytes, 96 bytes, 120 bytes, 144 bytes, 168 bytes, 192 bytes, 216 bytes, 240 bytes, and 264 bytes. This includes the outer IPV6 header as well as the IOAM overhead (Hop-by-hop containing 1 $\rightarrow$ 9 namespaces with a single option for three hops, and padding). The same behavior is observed on both figures, i.e., performance stability until five namespaces on `Merry`. In the fashion of previous experiment, we also exceed the free room in socket buffers. We observe a huge drop on `Sam` for 7, 8, and 9 namespaces. This can be explained by the fact that, in the IOAM dataplane, a namespace requires a lookup to check if it is known or not. Therefore, having a lot of namespaces per packet triggers multiple lookups. However, we do not think this is the only cause, as the hashtable is four or more times bigger than the maximum number of allowed namespaces, which reduces collisions. Except if the hash algorithm is weak, we should not observe such a high drop. Indeed, we see that it corresponds to the same drop on `Sam` for previous experiment, which happens with an approximatively 200-byte overhead. High is the chance that this is not a coincidence and there is also something else going on. This will be investigated deeper in the near future.

As an all-in-one example, we try a good compromise in a more realistic situation. This time, we use 800-byte packets which is approximatively the mean of our sub-experiments but could also well be the average size of, for instance, HTTP packets. The frequency of injections is 5%, with 5 namespaces each containing 3 options. While the same use case with a 100% frequency of injections would be too much, here we observe a loss of ~1.000.000 pps (~6.4 Gbps) from one end of the topology to the other, which represents ~16% of the traffic (~38.5 Gbps). Again, the use of IOAM may be specific to each, depending on the goal and conditions at a precise moment. It's up to operators to choose what suits them best.

All of those measurements are focused on the encapsulation use case. However, the encapsulation and direct insertion use cases are not so different. Indeed, the main difference between them is that, for the latter, it requires a bit more processing as the IPv6 header needs to be shifted in order to include the new Hop-by-hop containing IOAM headers. This is slower than the encapsulation process but it could also be faster if no re-allocation happens, which may be the case because fewer bytes are inserted.

Next Steps

The availability of IOAM inside the Linux Kernel source tree would be a huge asset. To this end, we will improve the current implementation in order for it to be accepted and merged. In parallel, now that IOAM works inside the Linux Kernel, we have already started combining LAYER5-7 tracing (OpenTelemetry) with IOAM tracing at network level, so that we end up with a complete LAYER2-7 tracing solution.

Conclusion

In-situ OAM (IOAM) allows operators, in a pre-defined domain, to insert telemetry data within packets without injecting additional traffic for measurements. As such, it has the potential to enhance operations. IOAM is still in its infancy as it is currently under standardization by the IETF. This paper reports what is, to the best of our knowledge, the very first implementation of IOAM for IPV6 in the Linux kernel[1] as well as early results on IOAM performance. We hope that our implementation supports IOAM standardization, drives adoption of IOAM as well as associated research. The current implementation will be improved in order to be merged inside the kernel. With the availability of IOAM for the Linux Kernel, we are now already evolving to a comprehensive tracing solution combining LAYER5-7 tracing (OpenTelemetry) with IOAM tracing at network level.

Acknowledgments

Mr. Iurman's work has been funded by a Cisco grant CG# 1343155.

References

[1] Bhandari, S.; Brockners, F.; Pignataro, C.; Gredler, H.; Leddy, J.; Youell, S.; Mizrahi, T.; A., K.; B., G.; Lapukhov, P.; M., S.; S., K.; and R., A. 2019. In-situ OAM ipv6 options. Internet Draft (Work in Progress) draft-ietf-ippm-ioam-ipv6-options-00.

[2] Brockners, F.; Bhandari, S.; Dara, S.; Pignataro, C.; Gredler, H.; Leddy, J.; Youell, S.; Mozes, D.; Mizrahi, T.; Lapukhov, P.; and Chang, R. 2017. Requirements for in-situ OAM. Internet Draft (Work in Progress) draft-brockners-inband-oam-requirements-03, Internet Engineering Task Force.

[3] Brockners, F.; Bhandari, S.; Pignataro, C.; Gredler, H.; Leddy, J.; Youell, S.; Mizrahi, T.; Mozes, D.; Lapukhov, P.; Chang, R.; and Bernier, D. L. J. 2019. Data fields for in-situ OAM. Internet Draft (Work in Progress) draft-ietf-ippm-ioam-data-08, Internet Engineering Task Force.

[4] Brockners, F.; Bhandari, S.; and Bernier, D. 2019. In-situ OAM deployment. Internet Draft (Work in Progress) draft-brockners-opsawg-ioam-deployment-00, Internet Engineering Task Force.

[5] Cisco. 2017. M-anycast. see `https://github.com/CiscoDevNet/iOAM/tree/master/M-Anycast`.

[6] Deering, S., and Hinden, R. 2017. Internet protocol, version 6 (ipv6) specification. RFC 8200, Internet Engineering Task Force.

[7] Facebook. fbtracert. see `https://github.com/facebook/fbtracert`.

[8] Facebook. Udppinger. see `https://github.com/facebook/UdpPinger`.

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

[10] Katz, D., and Ward, D. 2010. Bidirectional forwarding detection (BFD). RFC 5880, Internet Engineering Task Force.

[11] Kernel, L. HOWTO for the linux packet generator. `https://www.kernel.org/doc/Documentation/networking/pktgen.txt`.

[12] Li, Y.; Miao, R.; Liu, H.; Zhuang, Y.; Feng, F.; Tang, L.; Cao, Z.; Zhang, M.; Kelly, F. P.; and Alizadeh, M. 2019. HPCC: High precision congestion control. In *Proc. ACM SIGCOMM*.

[13] Mizrahi, T.; Sprecher, N.; Bellagamba, E.; and Weingarten, Y. 2014. An overview of operations, administration, and maintenance (OAM) tools. RFC 7276, Internet Engineering Task Force.

[14] OpenTelemetry. 2019. Vendor-neutral APISs and instrumentation for distributed tracing. `https://opentelemetry.io`.

[15] Postel, J. 1981. Internet protocol. RFC 791, Internet Engineering Task Force.

[16] V. Jacobson et al. 1989. traceroute. man page, UNIX. See source code: `ftp://ftp.ee.lbl.gov/traceroute.tar.gz`.

[17] Wieers, D. Dstat: Versatile resource statistics tool. `http://dag.wieers.com/home-made/dstat`.