Implementation of IOAM for IPv6 in the 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 the data packet, 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 propose a first preliminary evaluation of our implementation under a controlled environment. Our IOAM implementation is available as open source.

1 Introduction

Operations, Administration, and Maintenance (OAM) refers to a set of techniques and mechanisms for performing fault detection and isolation and for performance measurements. Multiple OAM tools have been developed throughout the years, for various layers in the protocol stack [1], going from basic traceroute [2] to Bidirectional Forwarding Detection (BFD [3]). Recently, OAM has been pushed further through In-Situ OAM (IOAM) [4, 5]. The term “In-Situ” directly refers to the fact that the OAM and telemetry data is carried within data packets rather than being sent within 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 [6] can be considered as an IOAM mechanism. However, compared to the Record-Route option, 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 information such as timestamps, sequence numbers, or even generic data such as queue size, 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 predefined path using a traffic steering method, such as service chaining (e.g., NFV service chaining) or traffic engineering (e.g., through Segment Routing [7]).

In a nutshell, IOAM gathers telemetry and operational information along a path, within the data packet, as part of an existing (possibly additional) header. It is encapsulated in IPv6 packets as an IPv6 HopByHop extension header [8]. 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 Tran-
This paper describes our implementation in Linux kernel 4.12.1 We first cover the IOAM node registration as well as any IOAM resources allocation (Sec. 2.1 and 2.2). We next explain how packet parsing can be done efficiently for IPv6 Extension Headers (Sec. 2.3). We finally focus on how IOAM headers are inserted and deleted from packets (Sec. 2.4 → 2.6). Our implementation is based on IOAM drafts [11, 5], respectively versions 02 and 05.

2 Linux-Kernel Implementation

In this section, we carefully describe how we have implemented IOAM within the Linux kernel 4.12.1 We designed our implementation to be as efficient as possible, through a new kernel module. This paper, also, discusses early results on performance of our IOAM implementation. In particular, we show that inserting IOAM headers and data does not strongly reduce the bandwidth capabilities of an IOAM domain but it comes with an additional (but reasonable) delay due to IOAM processing.

The remainder of this paper is organized as follows: Sec. 2 describes our open-source implementation of IOAM inside the Linux kernel; Sec. 3 evaluates the performance of our implementations with a few use cases; finally, Sec. 4 concludes this paper by summarizing its main achievements.

1Our 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

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{ioam_node}
\caption{IOAM node registration structure.}
\end{figure}
the same time. 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 node will ignore IOAM data from an unknown IOAM namespace. As for the list of IOAM insertions (encaps), it 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.

### 2.2 Node Registration

Calling the `ioctl` triggers 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 the node level as any 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 balanced between potential collisions and memory allocation. We achieve this by forcing the hashtable size being four times the maximum number of allowed IOAM namespaces. The IOAM node identifier is also stored along with the hashtable, as well as pre-allocated paddings, from length 1 to length 7, to speed up IOAM options alignment (see Sec. 2.4).

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. 2 illustrates how the buffer is structured and how it works (more details on this buffer in Sec. 2.4). It is allocated twice per IOAM encaps interface, one for a Hop-by-Hop option and another for a Destination option. The IOAM identifier and role are also stored there.

### 2.3 EH Parsing

IPv6 Extension Headers already come with parsing mechanisms inside the kernel, based on [12]. However, for speeding up as much as possible both IOAM header insertion and deletion, we need to maintain additional information. This is the objective pursued by the `ioam_parsed_eh` structure (see Fig. 3). It helps to easily insert and/or remove IOAM from packets, while remaining efficient. The usefulness of each field will be detailed in Sec. 2.4 and 2.6.

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 options 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
struct ioam_parsed_eh parsed_eh;
IP6CB(skb)->ioam = &parsed_eh;
if (ipv6_parse_hopopts(skb)){
    /* decap, if any, based on parsed_eh */
}
/* encap, if any, based on parsed_eh */
IP6CB(skb)->ioam = NULL;

Figure 4: Simplified example of IPv6 input for a Hop-by-Hop.

limit.

As both the header insertion and deletion need this data, we need to 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 use for 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 (4 or 8 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. 4. 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.

2.4 Header Insertion

The IOAM header insertion happens after both the parsing (Sec. 2.3) and the IOAM header deletion (Sec. 2.6). The first reason of such a decision is that we benefit from the parsing. It means that there is no need to do the same job again, and we also make sure that data is consistent and well-formed. The second reason is that we want to remove data first, if any, before inserting new ones. It allows for avoiding a possible allocation if the packet buffer has no free room anymore.

Two different cases can be met here. The first one is when there is no Hop-by-Hop option in the packet. The same logic can be applied to a Destination option. In that case, we simply insert the entire buffer shown in Fig. 2. The first two octets represent the EH header, while two next octets are padding. Indeed, IOAM options require a $4n$ alignment [5, 11], 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 [12]. This is the easiest and fastest case.

The other case is when there is a Hop-by-Hop (or Destination) option in the packet. In that case, we need to insert the IOAM data, but we first need to compute the head and tail padding required for both alignment and length. This is where the parsing data is useful, as illustrated in Fig. 5. The extra space to insert is computed based on new head and tail paddings, who themselves depend on the last padding found in the EH. We reuse the entire EH minus the tail padding, if any, and we append IOAM data with correct head and tail paddings to respect both alignment and length.
The strength of this algorithm is that it does not require any allocation, even for head and tail paddings, as they are already allocated at registration time. However, an implicit allocation can 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 encapsulation 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 done by the operator.

2.5 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 is required for a lookup inside the hashtable containing all known IOAM namespaces for the node. If those conditions are respected, the node inserts its data, depending on the asked IOAM mode/option. Some data may be unavailable inside the Linux kernel (e.g., queue depth, buffer occupancy).

Note that the data insertion also happens during IOAM header insertion, with the only difference that, here, the above checks are not applied. Indeed, it is the node itself that built the buffer. Any additional check in this situation would be useless and a clear waste of time.

2.6 Header Deletion

The IOAM header deletion is straightforward due to data provided by the improved parsing. We know the total size of data blocks to be removed as well as their respective positions and sizes, as shown in Fig. 3. One can distinguish two different cases. The first and fastest one is when the entire EH must be removed. It is possible to quickly check it since we know the size of data to be removed, as well as the total padding and the EH size. If decap_size + pad_size == EH.size - 2, then the entire EH must be removed (minus 2 to not take EH header into account). As for the second case, it will omit IOAM blocks that must be removed while shifting/copying data, and append tail padding if required.

Just like the header insertion algorithm, there is no allocation involved here. Of course, to stay efficient, options are not re-ordered so the arrangement could not be optimal. Still, it seems that alignment is still respected after the deletion, for any option that was positioned after a removed IOAM header, by only removing blocks without taking care of paddings before and after these blocks. At worst, alignment is respected with unnecessary paddings. An example is shown in Fig. 6a where an IOAM option (4n aligned) and another option (4n + 3 aligned) are present. Fig. 6b shows the result after the IOAM header deletion. One can see that the remaining option is still 4n + 3 aligned but 5-octet padding is present, where 1-octet padding would have been enough. Again, this is not a real problem as it will still work as expected, but this is not elegant. A possible solution would be to check if the IOAM header to be removed is in-between paddings and recompute the minimal merged padding size. However, it would have an impact on memory, as more data would need to be stored, as well as on performance. Consequently, it may not be worth for it, considering the usage and goal of IOAM.

3 Evaluation

In this section, we make a first attempt at evaluating IOAM performances inside the Linux kernel.
We first present our testbed (Sec. 3.1) and, next (Sec. 3.2), discuss our results.

3.1 Testbed

For performing the performance measurements, we setup a testbed on a single physical machine based on Intel Xeon CPU E5-2683 v4 at 2.10GHz, 16 Cores, 32 Threads, 64GB RAM, running Debian 9.0 with a 4.12 patched kernel to include our IOam implementation. Fig. 7 shows the five-node topology running on the machine, where each node is isolated inside a dedicated Linux container via lxc 2.0.7 [13].

Alpha is the traffic generator (TG), while Beta is the receiver. Three nodes represent the IOam domain whose boundaries are Athos and Aramis. They respectively insert and remove IOam headers. Porthos inserts its IOam data, just as both Athos and Aramis do too. Containers are linked through veth peers. The TG runs pktgen 2.75 [14]. pktgen is included in the kernel, meaning packets generated are directly handed by the network driver. pktgen is only able to generate UDP packets. Packets are sized to 1,200 bytes. This is due to the fact that some tests require insertions of ≈ 200 bytes in packets while we want to respect the MTU (default 1,500 bytes) and measure the same thing all along.

Each experiment lasts 60 seconds and is run 30 times. Each data point in subsequent plots represents the mean value over those 30 runs.

3.2 Results

We first examine the bandwidth baseline on a vanilla kernel, and compare it to the same kernel patched with IOAM. We distinguish two cases. Firstly, IOAM without registration means the IOAM module is enabled in the kernel but none of the nodes are registered. The goal is to measure the impact of the patch being totally passive. Secondly, IOAM with registration means that nodes are registered and, as such, IOAM capable. Fig. 8 shows the results for each measurement. We see a very slight decrease between a vanilla kernel (3.15 Gbps) and the patched IOAM kernel (3.13 Gbps). Unsurprisingly, the loss is more visible when nodes are registered (2.93 Gbps), although not that huge (a drop of 7%). In the remainder of this section, for subsequent tests, we only consider IOAM with registration on traffic that does not contain any HopByHop before entering the IOAM domain.

IOAM aims at inserting telemetry data inside IPv6 packet. 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% (i.e., no IOAM injection), 0.01%, 0.1%, 1%, 2%, 10%, 25%, 50%, and 100% (i.e., IOAM data is injected in every packet) of the packets traversing the IOAM domain. We consider a single namespace containing a single option. The impact on bandwidth is plotted on Fig. 9. Without any surprise, (i) the bandwidth reached with 0% of IOAM injection is consistent with results provided in Fig. 8 and (ii) the bandwidth decreases with proportion of packets involved in IOAM data injection. At worst (all packets traversing the
IOAM domain are involved in IOAM data injection),
the bandwidth is dropped by roughly 7%. However,
this extreme case is unrealistic as we do not expect
any operator inserting telemetry data in every packet
traversing its domain (the data processing load would
quickly become unmanageable). We rather expect
IOAM being applied on a tiny portion of the traffic (or
for very specific and limited in time use case), corre-
sponding so to results on the upper left of Fig. 9. In
such a case, the impact on the IOAM domain band-
width is quite limited (a reduction of 2% when in-
serting IOAM in 1% of the packets).

We next evaluate the effect of the number of IOAM
Trace options on the overall bandwidth of the IOAM
domain. Fig. 10 shows the results for 1 → 11 op-
tions. We inject IOAM data in every packets, corre-
sponding so to the extreme case discussed with Fig. 9
(the bandwidth for 1 option is thus equivalent to the
bandwidth achieved with 100% of packets impacted
by IOAM in Fig. 9). The bandwidth remains glob-
ally stable until six options are inserted and, then, it
starts marking a clear decrease. We believe that this
is due to the fact that inserting more options leads to
exceeding the free room in packets buffer. It therefore
requires a re-allocation of extra space by the kernel
to be able to deal with the entire IOAM data.

Fig. 11 investigates the impact of the number of
IOAM namespaces per packet on the overall band-
width of the IOAM domain. For those measurements,
each namespace contains a single IOAM Trace option
inserted in every packet (100% on Fig. 9), meaning
that the bandwidth achieved for a single namespace
corresponds to the one achieved for a single Trace op-
tion in Fig. 10. On Fig. 11, we can notice a decrease
between two and three IOAM namespaces inserted. In
the fashion of Trace option, we exceed here the free
room in packets buffer with three IOAM namespaces.
Thus, approximatively, three IOAM namespaces rep-
resent the same amount of bytes than six IOAM Trace
options. The bandwidth remains roughly stable un-
Figure 12: Additional delay due to IOAM operation in each node within the IOAM domain.

...five IOAM namespaces, where we observe another drop (attributed to the fact that there are more and more bytes to insert).

Finally, we estimate the additional delay due to IOAM processing in each node within the IOAM domain. Fig. 12 reports the minimum delays when varying the number of namespaces between one and seven. We first look at the encap node (i.e., “Athos”) and see three groups of additional delays: (1, 2) IOAM namespaces, (3, 4, 5) IOAM namespaces, and (6,7) IOAM namespaces. The increase in the delay between those groups is explained by the fact that we are adding more and more bytes in packets, requiring so more processing due to possible re-allocation. On the contrary, for the decap node (i.e., “Aramis”), the additional processing delay remains the same, whatever the number of namespaces considered. This is explained by the fact that the entire IPv6 HopByHop extension header is removed at once, no matter the number of namespaces inserted inside. The intermediate node (i.e., “Porthos”) encounters two groups of additional processing delay: (1,2,6,7) IOAM namespaces and (3,4,5) IOAM namespaces. We ascribe this behavior to some noise in measurements, as we expect the additional processing delay progressively growing with the number of namespaces considered (or, at least, be the same).

4 Conclusion

In-situ OAM (IOAM) allows operators, within a pre-defined domain, to insert telemetry data within data packet, without injecting additional traffic for measurements. As such, it has the potential to enhance operations. IOAM is still in its infancy as it is still 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 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. With the availability of IOAM for the Linux Kernel, we can evolve to a comprehensive tracing solution combining Layer 5-7 tracing (e.g. OpenTracing, OpenCensus) with IOAM tracing at network level.

References


