Linux Kernel Support for IOAM Direct Exporting

Maxime Goffart, Justin Iurman, Emilien Wansart, Benoit Donnet
Université de Liège – Montefiore Institute
Liège, Belgium
`{firstname}.{lastname}@uliege.be`

Abstract

In Situ **O**perations, **A**dministration, and **M**aintenance (IOAM) provides a way to collect telemetry data on devices, *e.g.*, switches and routers, in so-called limited domains (*e.g.*, a datacenter or a service provider). Recently standardized in the IETF, IOAM has been developed with various operation modes (*i.e.*, option-types) in order to be suitable for as many potential use cases as possible. One of the option-types, namely the **P**re-allocated **T**race **O**ption-Type (PTO), has already been implemented in the Linux kernel and is part of the mainline tree since version 5.15.

IOAM comes with another option-type, namely the IOAM **D**irect **EX**porting (DEX). It allows each network device on the path to locally aggregate and/or export telemetry data towards one or more collector(s). This paper proposes the very first implementation of IOAM DEX in the Linux kernel, with support in ubiquitous user space tools. The paper discusses our implementation and presents performance results of our implementation. Our source code is publicly available.

Keywords

Linux kernel, IOAM, Direct Exporting, Network Telemetry

Introduction

The last ten to fifteen years have witnessed a strong evolution of the Internet: from a hierarchical, relatively sparsely interconnected network to a flatter and much more densely interconnected network [11, 8, 24] in which hyper giant distribution networks (HGDNs, - *e.g.*, Meta, Google, or Netflix) are responsible for a large portion of the world traffic [4]. HGDNs are becoming the *de facto* main actors of the modern Internet. The very same set of actors have fueled the move to very large data center networks (DCNs).

In parallel, throughout the years, multiple *Operations, Administration, and Maintenance* (OAM) tools have been developed, for various layers in the protocol stack [17], going from basic `traceroute` to Bidirectional Forwarding Detection (BFD [15]) or recent UdpPinger [10] and Fbtracert [9]. The measurement techniques developed under the OAM framework have the potential for performing fault detection, isolation, and performance measurements.

Telemetry information (*e.g.*, timestamps, sequence numbers, or even generic data such as queue size and geolocation of the node that forwarded the packet) is key to HGDNs, DCNs, and Internet operators in order to tackle two particular challenges. First, the network infrastructure must be running all the time, even in the presence of (unavoidable) equipment failure, congestion, or change of traffic patterns. Second, customers want to enjoy their content in whatever context they access it (*e.g.*, at home on one or more device(s) or on a mobile device in public transportation) with the highest possible quality and the lowest delay. Consequently, HGDNs, DCNs, and classical Internet operators must carefully engineer their network to ensure the highest Quality-of-Experience (QoE) on the user side. Thus, network monitoring and measurements are of the highest importance for network operators [16, 23], although the available tools and methods [10, 9] have not kept up with the pace of growth in speed and complexity.

One of the new telemetry solutions under the OAM framework is **I**n Situ **O**perations, **A**dministration, and **M**aintenance (IOAM) [6], which is standardized by the IETF. IOAM extends existing data packets to collect operational and telemetry information inside a restricted network domain. It has many operation modes, referred to as option-types, that specify how the telemetry data is processed. IOAM can work on top of existing protocols such as IPV6 [3] or NSH [5].

In this paper, we present and evaluate the performance of our Linux kernel implementation of IOAM **D**irect **EX**porting (DEX) [19], which is one of the operation modes of IOAM. IOAM DEX triggers network nodes (*i.e.*, routers) on path to locally collect and/or export to one or more receiving entities rather than gathering telemetry data of each network node inside the packet header. Our implementation builds on the existing one for IOAM PTO, another IOAM option-type, which itself relies on the lightweight tunnel feature of the Linux kernel. As specified in RFC 9326 [19], our implementation supports the insertion of IOAM DEX into a subset of packets. Due to this feature, we are able to limit the impact on network performance for a limited injection rate as the subsequently presented results of the evaluation of our implementation highlight. For instance, for an injection rate of 1%, the maximal loss in performance, across all operations pertaining to IOAM DEX, is 7.49%. However, at an injection rate of 100% (*i.e.*, worst-case scenario, which should not be encountered in a realistic deployment), we reach a maximal loss of 66.59%.

Furthermore, we explain how we add support to some existing user space tools (*i.e.*, `iproute2` and `Wireshark`) to complement the kernel implementation. Finally, our work is open source for the community.

Background

IOAM

In Situ **O**perations, **A**dministration, and **M**aintenance (IOAM) [6] is an IETF proposed standard that provides a way to collect telemetry data on devices, *e.g.*, switches and routers, in so-called limited domains (*e.g.*, a datacenter or a service provider). In this context, such limited domains are called IOAM *domains*. IOAM cannot be considered as an active method because it relies on existing packets without generating new ones dedicated to telemetry. However, it needs to modify the existing packets to add pieces of information for collecting and processing telemetry data. Thus, it cannot be considered as a passive method either. Consequently, IOAM can be classified as an hybrid type 1 method based on the classification presented in RFC 7799 [18] because it is in-between a passive and an active method.

The INGRESS of an IOAM domain is called the encapsulating node, while the EGRESS is known as the decapsulating node. All IOAM nodes in-between are called transit nodes. For security reasons, the decapsulating node must be configured to properly remove any IOAM headers from packets to avoid leaking potentially sensitive telemetry information.

IOAM can be encapsulated in different network protocols [3, 5], such as IPV6 or NSH. In this paper, the implementation focuses on IOAM with IPV6, which uses an IPV6 Extension Header (either an IPV6 Hop-By-Hop or IPV6 Destination option) to carry the IOAM header. RFC 9197 [6] specifies different metrics (*i.e.*, IOAM data fields) that can be collected within an IOAM domain. For example, one can collect node and interface identifiers, timestamps, queue depths, etc. RFC 9197 also defines a list of operation modes (*i.e.*, *Option-Types*) as follows:

- **P**re-allocated **T**race **O**ption-Type (PTO). In this mode, the space to store the IOAM data fields of all the nodes on the path is pre-allocated by the encapsulating node. This mode has been implemented [14] and is available in the Linux kernel since version 5.15;
- Incremental Trace Option-Type. In this mode, each node extends the space before storing its IOAM data fields;
- **P**roof-**o**f-**T**ransit (PoT). This mode is used for proving that a packet followed a specific path;
- **E**dge-**to**-**E**dge (E2E). In this mode, transit nodes may process the IOAM data added by the encapsulating node but cannot modify it.

IOAM DEX

RFC 9326 [19] introduces a new IOAM Option-Type, called IOAM **D**irect **EX**porting (DEX). With this mode, each IOAM node along a path within an IOAM domain may trigger a local aggregation and/or export of its IOAM data, without adding IOAM data inside packets.

The IOAM DEX header is represented in Fig. 1. The `Namespace-ID` represents the IOAM namespace, which

0 15	16 23	24 31
Namespace-ID	Flags	Extension-Flags
IOAM-Trace-Type		Reserved
Flow ID (optional)		
Sequence Number (optional)		

Figure 1: IOAM DEX header.

is not to be confused with a Linux network namespace[1]. The bits inside the `Flags` byte are allocated by IANA [13]. However, none are currently standardized. The `Extension-Flags` is an 8-bit field where each bit represents the presence of an optional 4-byte field in the header. Currently, only the `Flow ID`, which is the flow identifier that can be used to identify packets belonging to a same flow, and the `Sequence Number`, which is an incremental counter of each packet in the flow, are specified. Finally, the `IOAM-Trace-Type` is a 24-bit field representing the expected IOAM data (see RFC 9197 [6]).

Fig. 2 illustrates how the IOAM **D**irect **EX**porting (DEX) works. The first hop, labeled `R1`, represents the encapsulating node and is responsible for inserting both the IOAM (`IOAM H`) and IOAM DEX (`DEX H`) headers between the packet header "H" and the packet payload "P". The encapsulating node may then locally aggregate and/or export its IOAM data, based on the `IOAM-Trace-Type` in the header. Similarly, transit hops respectively labeled `R2` and `R3`, and the last hop labeled `R4` representing the decapsulating node, may locally aggregate and/or export their IOAM data. Transit nodes cannot modify or remove the DEX header. Finally, the decapsulating node removes both the IOAM and IOAM DEX headers to avoid leaking data.

RFC 9326 [19] does not specify the functionality of local processing, or exporting methods and format. Thus, our kernel space implementation, which we present subsequently, is intentionally as generic as possible in order to leave these decisions to the network operators that may want to use IOAM DEX. However, we propose a user space solution, which we detail later on, for exporting the IOAM data under the **IP F**low **I**nformation e**X**port (IPFIX) protocol [2]. IPFIX is a protocol for exporting flow information from routers inside a network. This proposed solution is inspired by a work in progress in the IETF. An expired draft [20] suggesting this approach will soon be replaced by a new one.

Kernel Space Implementation

As previously mentioned, one of the operation modes (*i.e.*, option-types) of IOAM, namely **P**re-allocated **T**race **O**ption-Type (PTO), has already been implemented in the Linux kernel [14]. However, the version described in the referenced paper is an old one and does not reflect the current implementation. In this paper, we focus on the implementation of **D**irect **EX**porting (DEX) by relying on the existing one while ensuring backward compatibility.

[1]Note that the existing implementation for the PTO and the new one for the DEX are both compatible with Linux network namespaces.

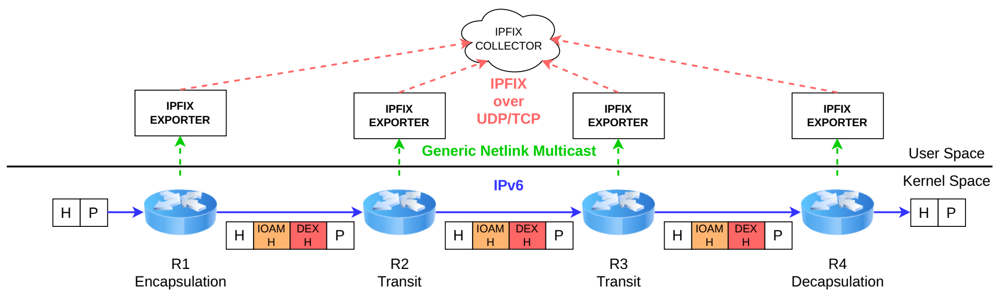

Figure 2: Packet across an IOAM domain using IOAM DEX with the proposed exporting of IOAM data under the IPFIX format [2].

Encapsulation

For encapsulating the IOAM DEX header, we enhance the current implementation for the IOAM PTO option-type that relies on the lightweight tunnel API. It allows one to attach the tunnel attributes to the route as depicted in Listing 4.

The data structure representing the tunnel state (`ioam6_lwt`) has been extended with a reference to a new structure, named `ioam6_lwt_dex` and depicted in Listing 1, containing the IPV6 Hop-By-Hop option (`struct ipv6_hopopt_hdr`) with both IOAM (`struct ioam6_hdr`) and IOAM DEX (`struct ioam6_dex_hdr`) headers. Doing so, we ensure backward compatibility with the PTO while leading to the minimal number of modifications to existing code. The structure `ioam6_dex_hdr` is the IOAM DEX header as depicted in Fig. 1. Similarly, the function initializing the tunnel state (`ioam6_build_state`), which is called when the route is configured from user space with `iproute2` (more details in the following section), has been updated to support the parameters required for the DEX and to store them in the new structure.

```
struct ioam6_lwt_dex {
   struct ipv6_hopopt_hdr eh;
   u8 pad[2]; /* 2-octet padding for 4n-alignment */
   struct ioam6_hdr ioamh;
   struct ioam6_dex_hdr dexh;
} __packed;
```

Listing 1: `ioam6_lwt_dex`.

As detailed previously, the IOAM DEX header can contain optional data fields depending on the value of the 8-bit field `Extension-Flags` (see Fig. 1). Currently, two such fields are standardized by IANA, namely `Flow ID` and `Sequence Number`. However, the RFC 9326 [19] does not specify how to calculate the flow ID. We decided to compute the flow ID similarly to how the flow label is calculated in the implementation of Segment Routing with IPV6 as forwarding plane (SRv6) in the kernel[2] with the function represented in Listing 2.

[2]See function `seg6_make_flowlabel` in `seg6_iptunnel.c`.

```
static __be32 ioam6_dex_flowid(struct sk_buff *skb)
{
   u32 hash = skb_get_hash(skb);
   hash = rol32(hash, 16);
   return (__force __be32)hash;
}
```

Listing 2: Computation of flow ID based on packet.

Then, in order to store the mappings between the flow ID and the current value of the sequence number, we added an extra resizable hash table, named `dex_flows`, to the structure `ioam6_pernet_data` (see Listing 3) saving the IOAM data per Linux network namespace. The flow ID is used as the key while the sequence number is the stored value.

```
struct ioam6_pernet_data {
   struct mutex lock;
   struct rhashtable namesapces;
   struct rhashtable schemas;
   struct rhashtable dex_flows;
};
```

Listing 3: `ioam6_pernet_data`.

We decided to enable the `automatic shrinking` option in the parameters of the resizable hash table, represented by the structure `rhashtable_params`[3], that enables the dynamic and automatic reduction in size of the hash table. In our code, we do not remove any element from the hash table. Thus, the automatic shrinking feature does not change the performance. We have validated this hypothesis by evaluating the performance when automatic shrinking is disabled and the results are marginally the same whether the automatic shrinking is enabled or not.

Afterwards, packets going through the created lightweight tunnel, based on route configuration, will have the IPV6 DEX header as an IPV6 Hop-By-Hop Extension Header by going through the updated `ioam6_output` function, which uses `ioam6_do_inline` or `ioam6_do_encap` depending on whether the encapsulation is in inline (*i.e.*, modification of the IPV6 header) or encapsulation (*i.e.*, leading to the creation of an IPV6-in-IPV6 tunnel) mode. The two aforementioned functions are used for both the PTO and DEX, but they are

[3]In file `include/linux/rhashtable-types.h`.

capable to add the appropriate header based on the route configuration.

Finally, for every packet entering the lightweight tunnel, the kernel will trigger, using the method `ioam6_event`, a Generic Netlink [21] multicast event, as depicted in Fig. 2, containing the values inside the IOAM DEX header combined with IOAM data depending on the `IOAM-Trace-Type` field in the header. We chose to report the IOAM data using Generic Netlink to have a solution as generic as possible that leaves the decision on the inner workings of the local processing and/or exporting to end users since the RFC does not specify how to do so.

Transit

The function, namely `ipv6_hop_ioam`, handling in-transit packets containing IOAM headers as IPV6 Hop-By-Hop options has been modified to select the code path depending on the IOAM option-type, being either PTO (option-type 0) or DEX (option-type 4). If IOAM is enabled on the input interface and an IOAM namespace is configured on the node with a value matching the one in the DEX header, the same function, namely `ioam6_event`, as for the encapsulation code path will be called to generate a Generic Netlink multicast event containing the values in the IOAM DEX header with IOAM data. The types of the Generic Netlink attributes are described in the `ioam6_genl.h uapi`.

Decapsulation

For the decapsulation process, two operations need to be performed. First, the IOAM data must be reported using Generic Netlink. This is performed in the same code path as for the encapsulation and transit operations using the `ioam6_event` function with the same conditions (*i.e.*, IOAM enabled on input interface and configured IOAM namespace). Then, both IOAM and IOAM DEX headers must be removed to prevent the leakage of potentially sensitive telemetry information. If the destination of the packet is the decapsulating node and the encapsulation was done in inline node (*i.e.*, IOAM headers attached to the existing IPV6 header), the packet has reached its destination and will be handled by the existing receiving function of the kernel without being forwarded. If the destination of the packet is not the decapsulating node and the encapsulation was done in encapsulation mode (*i.e.*, leading to the creation of an IPV6-in-IPV6 tunnel), both IOAM headers will be removed by existing code in the kernel since the headers are attached to the outer IPV6 header having the decapsulating node as its destination.

User APIs

Three user APIs (`uapi`) have been updated, which will be used in `iproute2` to interact with the kernel space implementation.

First, the structure `ioam6_dex_hdr` representing the IOAM DEX header has been added to the `uapi ioam6.h`.

Then, `ioam6_iptunnel.h`, used for the configuration of the lightweight tunnel, is updated by adding an enumeration representing the IOAM option-type being used. Additionally, another constant is added in the enumeration representing the data provided when creating the tunnel for specifying the content of the IOAM DEX header.

Finally, a constant for IOAM DEX has been added to the enumeration representing the type of IOAM Generic Netlink multicast event with new Generic Netlink attribute types for IOAM DEX in the `uapi ioam6_genl.h`.

No breaking change has been done to the user APIs in order to ensure backward compatibility. So, existing tools will be functional, without any modification, on the patched version of the kernel.

User Space Implementation

For facilitating the usage of our kernel space implementation, we updated two existing user space tools (*i.e.*, `iproute2` and `Wireshark`) and created a new one to convert IOAM telemetry data to IPFIX [2, 20].

Route Configuration with `iproute2`

In order to append the IOAM DEX header as an IPV6 Extension Header using Linux's lightweight tunnel API, the encapsulating node needs to configure the route using the Linux routing socket (`rtnetlink`). It relies on the updated `ioam6_iptunnel.h uapi`. To facilitate the configuration of the route, we extended the support for IOAM in `iproute2`, which was introduced for supporting the encapsulation of the IOAM PTO, by adding the possibility to configure a route using IOAM **D**irect **EX**porting (DEX) with the following syntax:

```
$> ip -6 route add {} encap ioam6 [freq {}/{}] [mode inline | encap
      | auto] [tundst {}] dex ns {} trace-type {} ext-flags {} via
     {}
```

Listing 4: Adding route with IOAM DEX.

The values for the IOAM namespace[4] (`ns`), extension flags (`ext-flags`), and `trace-type` will be used to fill the header, which is depicted in Fig. 1. `iproute2` does not take the `flags` as a parameter since, at the time of writing, none are currently defined and the byte in the header will be set to 0 by the kernel space implementation.

The provided parameters will be retained in kernel space inside the state of the created lightweight tunnel for appending the IOAM DEX header in subsequent packets.

Node Configuration

Every router on the path, including both encapsulating and decapsulating nodes, must be configured for the processing related to IOAM DEX to happen. We rely on the following settings that were implemented for the addition of the IOAM PTO to the kernel [14].

First, each node must enable IOAM and configure their IOAM node identifier, which can either be the same for every interface or per-interface dependent. For both these settings, the network operators can configure them using `sysctl` to

[4]Remind that an IOAM namespace and a Linux (network) namespace are two different concepts that must not be confused.

modify the kernel parameters, which are the following where `{iface}` is the name of an existing network interface[5]:

- `net.ipv6.ioam6_id`;
- `net.ipv6.ioam6_id_wide`;
- `net.ipv6.conf.{iface}.ioam6_enabled`;
- `net.ipv6.conf.{iface}.ioam6_id`;
- `net.ipv6.conf.{iface}.ioam6_id_wide`.

Furthermore, the IOAM namespace must be configured on the nodes. This configuration relies on `iproute2`, which uses the updated `ioam6_genl.h uapi`, with the following command:

```
$> ip ioam namespace add ID [data DATA32] [data DATA64]
```

Listing 5: Configuring an IOAM namespace.

If, on any IOAM node in the domain, the namespace is configured and IOAM is enabled on the input interface, the kernel of the node will generate a Generic Netlink multicast event for every packet containing the IOAM DEX header under the condition that the value of one of the configured namespaces on the node matches the `ns` value in the header (see Fig. 1).

`Wireshark`

`Wireshark` [22] is the leading software for capturing and analyzing network packets. It supports a continuously increasing amount of protocols thanks to contributions by countless developers. Due to the prevalence of `Wireshark`, we decided to contribute to it by enhancing the IPV6 dissector, allowing one to properly inspect IPV6 packets, to support the decoding of the IOAM DEX header.

Our contribution [12] has been merged in the mainline repository. Yet, at the time of writing this paper, a new version has not been released. Thus, one needs to compile `Wireshark` from the sources to benefit from our patch.

IPFIX Exporter

As mentioned earlier, RFC 9326 [19] does not specify the inner workings of local processing nor the exporting methods and format. Thus, as explained in the presentation of our kernel space implementation, we trigger a Generic Netlink multicast event for every packet with the IOAM DEX header under some aforementioned conditions. This allows our solution to be as generic as possible and let end-users decide how to process and/or export the telemetry data.

Yet, we propose a solution, as depicted in Fig. 2, which relies on a *IPFIX exporter*. This exporter is a `Golang` application that listens for Generic Netlink events related to IOAM DEX and converts them to IPFIX. Furthermore, it can display the received events and/or send them to an IPFIX collector over `UDP` depending on the given parameters. We open-source this piece of software with our implementation of IOAM DEX.

[5]The interface can be `all` for all the interfaces or `default` for setting a default value.

Evaluation

Methodology

To evaluate the performance of IOAM DEX, we built a testbed made of two servers. The first one is used as a traffic generator, using TREX [7], while the second one served as a **D**evice **U**nder **T**est (DUT). The characteristics of both machines are summarized in Table 1. Both servers are connected with two direct attach copper cables through an Intel XL710 network interface card equipped with two ports, each capable of 40Gb/s. The DUT is configured to maximize the performance (*e.g.*, CPU in performance mode and network optimized settings).

Metric	Traffic Generator	DUT
RAM	32GB DDR4	16GB DDR4
Kernel	Linux 5.17	Patched Linux 6.12
CPU	Intel Xeon E5-2630v3 (8c/16t - 2.4GHz/3.2GHz)	

Table 1: Specification of machines used for the evaluation.

In this section, the term "baseline" refers to the maximum number of IPV6 packets per second (pps) that our DUT can forward. The value of the baseline is 1.07 million pps per CPU core (equivalent to 12.84Gb/s per CPU core)[6]. For comparison purposes, the baseline packets are IPV6 packets, without IOAM headers, of the same size (*i.e.*, 1,500 bytes, which is the default MTU in Linux) as the IOAM enabled packets. Abdelsalam et al. [1] obtained 1.119 million packets per second with a similar setup, confirming so our results.

We intentionally focused on a single flow on a single CPU core to establish a stable comparison basis. The DUT is configured to use a single queue for all incoming traffic with a single core responsible for that queue. This setup allows us to accurately measure the proportional impact of IOAM DEX. Scaling to multiple flows across multiple cores would have partially hidden our evaluation objective and can be deduced from our basis.

Results

We measured the number of packets per second processed by the DUT when performing the three operations pertaining to IOAM DEX independently (*i.e.*, encapsulation, transit, and decapsulation) depending on the percentage of packets being expanded with or containing the IOAM DEX header because the percentage of injection to use in a real-world deployment is operator and scenario-dependent.

We repeated each experiment 10 times, each one lasted for 30 seconds, and calculated the mean. On the subsequent figures, the standard deviations are also depicted. Yet, there are too small to be distinguishable.

Encapsulation We decided to evaluate the performance when the DUT is acting as the encapsulating node depending on three variables: (i) the IOAM trace-type representing the IOAM data to be collected and/or exported; (ii) the extension-flags for the optional inclusion of the flow ID and/or

[6]We were able to reach the line rate (*i.e.*, 40Gb/s) using four flows distributed across four CPU cores.

sequence number; (iii) the mode of IOAM being either inline (*i.e.*, modification of existing IPV6 header) or encapsulation (*i.e.*, IOAM DEX header in an additional IPV6 header leading to the creation of an IPV6-in-IPV6 tunnel) with or without specifying the source of the tunnel.

As depicted in Fig. 3, the IOAM trace-type has no impact on the encapsulation performance for a given injection rate. This is an expected result since the size of the IOAM DEX header, as represented in Fig. 1, does not depend on the trace-type and all the IOAM data fields have a size of either 4 or 8 bytes, so the impact on the reporting of IOAM telemetry with Generic Netlink is negligible.

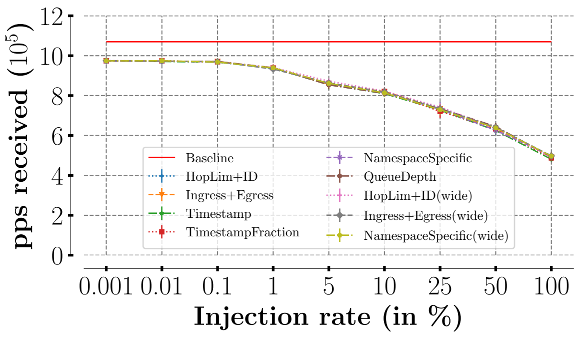

Figure 3: IOAM DEX encapsulation depending on IOAM trace-type.

When using an injection rate greater or equal to 10%, not using extension-flags provides a marginally better encapsulation performance compared to using them as represented in Fig. 4. In general, using extension-flags requires to compute the flow ID based on the packet, a hash table lookup to verify if the flow is already known by the kernel, and potentially a hash table insertion if it is a new flow. Thus, the only difference between using only the flow ID or both the flow ID and the sequence number lies in the appending of one or two 4-byte integer(s) in the header. So, it justifies the lack of difference in performance between the usage of one or two extension-flag(s) and the marginal benefit of not using extension-flags.

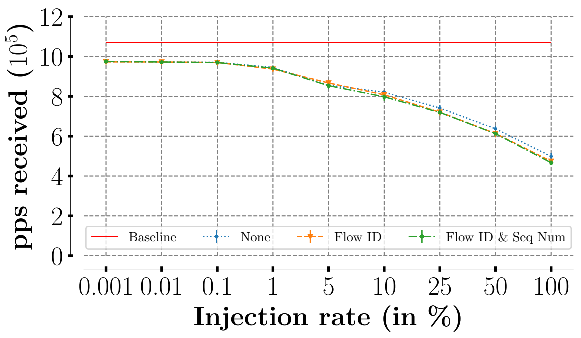

Figure 4: IOAM DEX encapsulation depending on DEX extension-flags.

However, there is a clear difference in performance depending on whether the source of the tunnel for the encapsulation mode (*i.e.*, leading to the creation of an IPV6-in-IPV6 tunnel) is specified from user space as pictured in Fig. 5. If the source of the tunnel is not given (orange line in Fig. 5), the kernel needs to perform a dynamic resolution to determine the source IPV6 address of the new header, which is a costly operation. We reached the same conclusion during the evaluation of the implementation of the IOAM PTO in the kernel. The uplift in performance by specifying the source of the tunnel compared to dynamic resolution ranges from 3.3% at a 5% injection rate to 33.8% at a 100% injection rate.

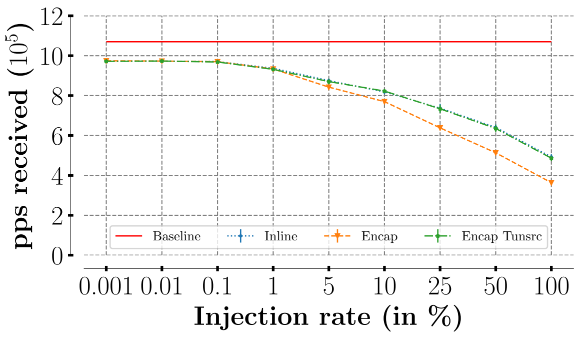

Figure 5: IOAM DEX encapsulation depending on IOAM encapsulation type.

Transit We decided to evaluate the performance when the DUT is acting as the transit node (*i.e.*, exporting IOAM telemetry data) in inline mode depending on two variables: (i) the IOAM trace-type representing the IOAM data to be collected and/or exported; (ii) the extension-flags for the optional inclusion of the flow ID and/or sequence number.

For the transit depending on the IOAM trace-type, based on Fig. 6, we reached the same conclusion than for the encapsulation, *i.e.*, the trace-type does not impact the performance for a given injection rate. However, for an injection rate less than 25%, the performance for the transit is better than the encapsulation because the transit node does not need to add an IPV6 Hop-By-Hop option to the packet.

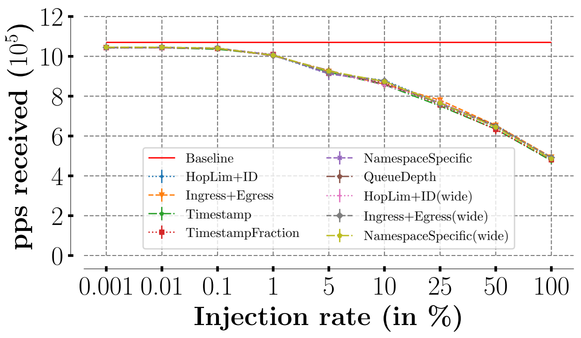

Figure 6: IOAM DEX transit depending on IOAM trace-type.

Similarly, the optional usage of extension-flags does not impact the transit performance for a given injection rate, as depicted in Fig. 7, since it pertains to copying zero, one, or two 4-byte integer(s) to the triggered Generic Netlink message. Additionally, as for the IOAM trace-type and injection rate less than 25%, the transit exhibits better performance than the encapsulation because it only needs to trigger a Generic Netlink event.

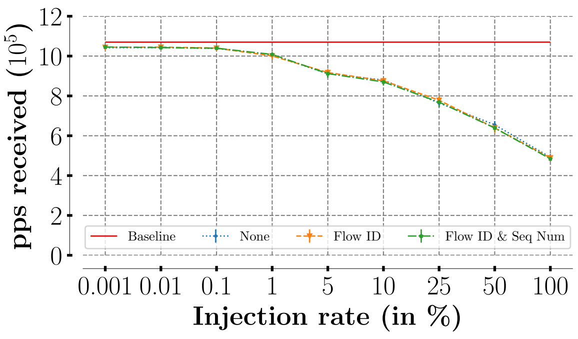

Figure 7: IOAM DEX transit depending on DEX extension-flags.

Decapsulation As for the transit, we decided to evaluate the performance when the DUT is acting as the decapsulating node (*i.e.*, exporting IOAM telemetry data and removing both IOAM and IOAM DEX headers) depending on two variables: (i) the IOAM trace-type representing the IOAM data to be collected and/or exported; (ii) the extension-flags for the optional inclusion of the flow ID and/or sequence number.

Due to our methodology, it is only possible to accurately measure the decapsulation operation when the IOAM DEX header is in encapsulation mode (*i.e.*, IPV6-in-IPV6 tunnel).

Once again, we reach the conclusion that the IOAM trace-type has no impact on the decapsulation performance for a given injection rate as depicted in Fig. 8. This is expected since the decapsulating node needs to report the IOAM data over Generic Netlink as the encapsulating and transit nodes (*i.e.*, same code path in the kernel). Then, the removal of the additional IPV6 header inserted by the creation of the tunnel is the same as for any tunnel in the kernel without any operations dedicated to IOAM DEX. This additional operation leads to worse performance compared to the transit starting from an injection rate of 5%.

The extension-flags do not impact the performance of the decapsulating node as represented in Fig. 9. However, it is more costly to perform the decapsulation compared to the transit (see Fig. 7) starting from an injection rate of 10% because, additionally to the reporting of IOAM data over Generic Netlink as for the transit node, the decapsulating node also needs to remove the additional IPV6 header induced by the usage of the IPV6-in-IPV6 tunnel.

Conclusion

IOAM **D**irect **EX**porting (DEX) lets routers along a path collect telemetry data related to in-transit packets. Then, this

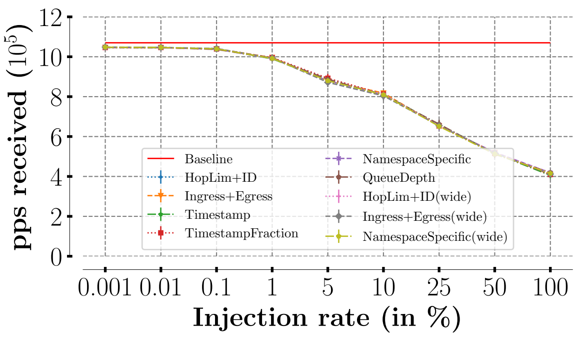

Figure 8: IOAM DEX decapsulation depending on IOAM trace-type.

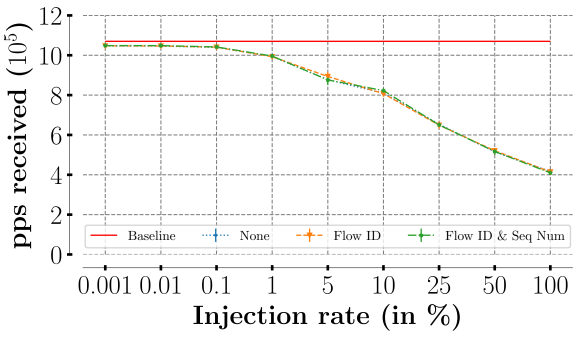

Figure 9: IOAM DEX decapsulation depending on DEX extension-flags.

data can be locally aggregated and/or exported depending on the needs of the network operators.

In this paper, we presented a kernel space implementation of IOAM DEX that does not make any assumption on the processing and/or exporting of the telemetry data. However, we propose and provide the implementation for a solution that exports the telemetry data towards an IPFIX collector by supplying an exporter that converts the IOAM data received from the kernel to IPFIX format. Additionally, our kernel space implementation does not brake any existing implementation to ensure full backward compatibility. Finally, we provide support for well-known user space tools (*i.e.*, `iproute2` and `Wireshark`) to facilitate the adoption of our kernel space implementation.

In the future, we hope to get some feedback on our implementation in order to create a new version suitable for being integrated in the mainline Linux kernel repository.

Source code

The repository gathering the source code for both kernel and user space implementations described in this paper, as well as an explanation on how it works, is freely available at the following URL: `https://github.com/Advanced-Observability/ioam-direct-exporting`

Acknowledgments

This work has been funded by the CyberExcellence project funded by the Walloon Region of Belgium, under number 211018, and the Feder CyberGalaxia project.

References

[1] Abdelsalam, A.; Ventre, P. L.; Mayer, A.; Salsano, S.; Camarillo, P.; Clad, F.; and Filsfils, C. 2018. Performance of IPv6 segment routing in linux kernel. In *Proc. International Conference on Network and Service Management*.

[2] Aitken, P.; Claise, B.; and Trammell, B. 2013. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. RFC 7011, Internet Engineering Task Force.

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

[4] Böttger, T.; Cuadrado, F.; Tyson, G.; Castro, I.; and Uhlig, S. 2018. Open Connect Everywhere: A Glimpse at the Internet Ecosystem Through the Lens of the Netflix CDN. *ACM SIGCOMM Computer Communication Review* 48(1).

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

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

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

[8] Dhamdhere, A., and Dovrolis, C. 2010. The Internet is Flat: Modeling the Transition from a Transit Hierarchy to a Peering mesh. In *Proc. ACM CoNEXT*.

[9] Facebook. 2024a. fbtracert. [Last Accessed: June 6th, 2024].

[10] Facebook. 2024b. UdpPinger. [Last Accessed: June 6th, 2024].

[11] Gill, P.; Arlitt, M.; Li, Z.; and Mahant, A. 2008. The Flattening Internet Topology: Natural Evolution, Unsightly Barnacles or Contrived Collapse? In *Proc. Passive and Active Measurement Conference (PAM)*.

[12] Goffart, M. 2024. Support for Ioam Direct Export in Wireshark. [Last Accessed: November 18th, 2024].

[13] IANA. 2021. In Situ OAM (IOAM). [Last Accessed: May 3nd, 2024].

[14] Iurman, J.; Donnet, B.; and Brockners, F. 2020. Implementation of IPv6 IOAM in Linux Kernel. In *Proc. Technical Conference on Linux Networking (Netdev 0x14)*.

[15] Katz, D., and Ward, D. 2010. Bidirectional Forwarding Detection (BFD). RFC 5880, Internet Engineering Task Force.

[16] Khemani, R. 2021. Advanced telemetry is a requirement in modern data center networks. [Last Accessed: June 6th, 2024].

[17] 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.

[18] Morton, A. 2016. Active and Passive Metrics and Methods (with Hybrid Types In-Between). RFC 7799, Internet Engineering Task Force.

[19] 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.

[20] Spiegel, M.; Brockners, F.; Bhandari, S.; and Sivakolundu, R. 2024. In Situ OAM Raw Data Export with IPFIX. Internet Draft (Work in Progress) draft-spiegel-ippm-ioam-rawexport-07, Internet Engineering Task Force.

[21] The Linux Foundation. 2017. Generic Netlink Howto. [Last Accessed: May 2nd, 2024].

[22] Wireshark Foundation. 2024. Wireshark. [Last Accessed: November 18th, 2024].

[23] Yu, M. 2019. Network telemetry: Towards a top-down approach. *ACM SIGCOMM Computer Communication Review* 49(1):11–17.

[24] Zhao, H., and Bi, J. 2013. Characterizing and Analysis of the Flattening Internet Topology. In *Proc. International Symposium on Computers and Communications (ISCC)*.