Securing IOAM in the Linux Kernel: Toward Trustworthy In Situ Network Telemetry

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

Abstract

In Situ Operations, Administration, and Maintenance (IOAM) is an IETF-standardized in-band network telemetry protocol that enables routers to collect and embed operational telemetry data directly into IPv6 Extension Headers of in-transit packets. IOAM is designed to operate within a Limited Domain — such as an Internet Service Provider (ISP) or a datacenter network — where boundary filtering is assumed to prevent telemetry data from leaking outside the domain. However, IOAM provides no built-in confidentiality or integrity protection: telemetry fields are transmitted in plaintext and are not authenticated, leaving them vulnerable to interception and forgery by on-path adversaries in the event of a misconfiguration or boundary enforcement failure.

To address this gap, we propose a security mechanism providing encryption and authentication for IOAM based on an AEAD scheme, supporting both AES-GCM and ChaCha20-Poly1305. We implement this solution directly in the Linux kernel, with a user-space configuration interface, and evaluate its impact on IPv6 packet forwarding performance. Both the kernel-space and user-space code are released as open source.

Keywords

Linux kernel, IOAM, Network Telemetry, Security

Introduction

In-band network telemetry protocols such as In Situ Operations, Administration, and Maintenance (IOAM) [5], INT [20], Alternate-Marking [10, 11], and Active Network Telemetry [28] have been developed and standardized to monitor complex network infrastructures [37, 35]. Rather than relying on dedicated probe traffic, these protocols embed telemetry data directly into the headers of user packets as they traverse the network, reducing measurement overhead and improving collection efficiency.

IOAM in particular operates within a *Limited Domain* [6], i.e., an isolated and self-managed network environment protected by strict boundary filtering. Within such a domain, IOAM nodes encapsulate, insert, and decapsulate telemetry data carried in IPv6 Extension Headers [8, 1]. However, IOAM provides no built-in mechanism to ensure the confidentiality or integrity of this data: telemetry fields are transmitted in plaintext and are not authenticated, leaving them exposed to interception and forgery in the event of a misconfiguration or boundary enforcement failure. The IOAM data fields may contain sensitive information, such as node identifiers, timestamps, and performance metrics (e.g., "Queue Depth" and "Buffer Occupancy" fields). If an adversary can access or modify this data, they could gain insights into the network's structure and behavior, or even inject false telemetry data to mislead network operators. A list of all the available data fields is provided in the appendices.

In this paper, we address this security gap by designing, implementing, and evaluating a security mechanism for IOAM directly in the Linux kernel. Our solution applies an AEAD scheme — either AES-GCM or ChaCha20-Poly1305 — to encrypt and authenticate IOAM data fields carried in IPv6 Extension Headers, providing strong confidentiality and integrity guarantees with no changes to the IOAM specification. Our main contributions are as follows:

- We propose a practical IOAM security solution based on AEAD, supporting both AES-GCM and ChaCha20-Poly1305;
- We implement this solution in the Linux kernel and release the full kernel-space and user-space code as open source;
- We evaluate the performance impact of this solution on IPv6 packet forwarding, demonstrating its practical viability.

The remainder of this paper is structured as follows. First, we provide the necessary background on both IOAM and AEAD. We then discuss the IOAM threat model. Afterwards, we present the proposed security solution and its implementation in the Linux kernel. We also explain how to set up and configure our secure version of IOAM on a Linux host. Next, we evaluate the performance of the kernel implementation and position this work with respect to the state of the art. Finally, we conclude the paper and discuss potential future work.

Background

This section provides the background required for the remainder of this paper. First, it gives an overview of IOAM. Then, it discusses AEAD.

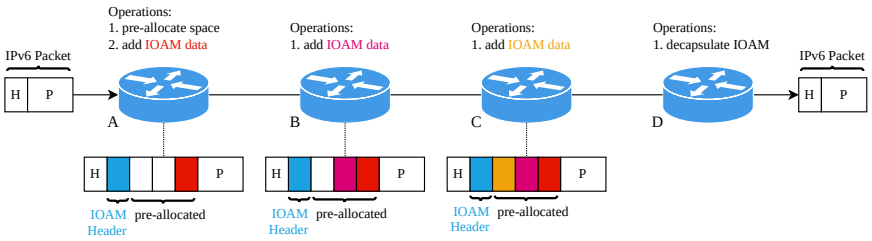

Figure 1: Generic example of IOAM data insertion. The symbol "H" denotes the IPv6 header, while "P" represents the IPv6 payload.

IOAM

In Situ Operations, Administration, and Maintenance (IOAM) is a network telemetry protocol standardized by the IETF [5]. It is designed to collect telemetry data within a Limited Domain, as defined in RFC 8799 [6]. Examples of such Limited Domains include Internet Service Providers (ISPs) and Data-Center Networks (DCNs). The term "In Situ" indicates that IOAM does not rely on dedicated telemetry packets. Instead, it uses user data packets and can be encapsulated in other protocols, including IPv6 [1], through a Hop-By-Hop option, as depicted in Fig. 6, and Network Service Header (NSH) [3].

Many operation modes, referred to as *option-types*, have been standardized to support a wide range of use cases. Currently, only Pre-allocated Trace Option-type (PTO) [5] and Direct EXporting (DEX) [34] have been implemented in the Linux kernel [18, 12] and publicly released to the community. To date, no IETF-compliant hardware implementation has been publicly released.

In this paper, we focus on the PTO carried over IPv6, as it is available in the Linux kernel and can be easily modified. It operates as follows. First, an *encapsulating* node (*i.e.*, ingress of the domain, denoted as node A in Fig. 1) inserts the IPv6 Extension Header with a single Hop-By-Hop option. This option contains the pre-allocated data space for subsequent nodes, as depicted in Fig. 6. Optionally, the encapsulating node can also add its own telemetry data inside the PTO. Then, *transit* nodes (denoted as nodes B and C in Fig. 1) add their respective telemetry data to the header space pre-allocated by the encapsulating node. Finally, the *decapsulating* node (*i.e.*, egress of the domain, denoted as node D in Fig. 1) removes the IPv6 Extension Header from the in-transit data packets. The removed header may optionally be sent to a telemetry collector for storage of the telemetry data.

IOAM telemetry data consists of IOAM data fields [5], each of which can be enabled or disabled individually through the `IOAM Trace-Type` field, as shown in Fig. 6. The telemetry data is added in plaintext inside the IPv6 Extension Header.

AEAD

Authenticated Encryption with Associated Data (AEAD) [32] is a cryptographic construction that provides both confidentiality and authenticity using a secret key. Compared to Authenticated Encryption (AE) [2], it supports Associated Data[1], which is authenticated but not encrypted. In an AEAD scheme, the plaintext payload is encrypted into a ciphertext, and an authentication tag is generated to protect both the ciphertext and the Associated Data.

Many AEAD schemes have been standardized and deployed, including the Galois/Counter Mode (GCM), a mode of operation for block ciphers [24, 9] used with AES (*i.e.*, AES-GCM), and ChaCha20-Poly1305 [26, 27]. Both schemes are supported in Transport Layer Security (TLS) since version 1.2 [33, 23].

AEAD is well-suited for securing network packets. The payload is both encrypted and authenticated, whereas the header is only authenticated (*i.e.*, used as Associated Data). This allows middleboxes (*e.g.*, routers) to process the header while guaranteeing that the payload cannot be interpreted or modified in transit.

A critical requirement of any AEAD scheme is nonce uniqueness: the nonce passed to the encryption function must never be reused under the same key [25, Sec. 5.1.1]. Doing so would compromise both the confidentiality of the encrypted data, the authenticity, and integrity guarantees provided by the scheme [25], a requirement reiterated in NIST SP 800-38D [9] for GCM and in RFC 8439 [27] for ChaCha20-Poly1305.

Our implementation addresses this requirement by using a monotonically incrementing nonce, combined with a user-configurable key update mechanism to prevent nonce exhaustion under a single key. The design and implementation details are presented in the following sections.

Threat Model

As introduced in Sec. "Background", IOAM operates within a Limited Domain whose boundary routers represent the *trust boundary* between the trusted network and the untrusted Internet. Within this trust boundary, the *assets* are the network packets carrying sensitive IOAM telemetry data and the centralized collector or collectors storing the telemetry data of the routers. The *weaknesses* of IOAM arise from

[1] In the literature, Additional Authenticated Data (AAD) is used interchangeably with Associated Data (AD).

the lack of both encryption and authentication in the IOAM specification and implementation.

We consider two types of adversaries: *external attackers*, located outside the Limited Domain, and *internal attackers*, located within it. Both adversary types can perform similar attacks, but from different vantage points.

The first attack scenario is *telemetry injection*: an adversary inserts misleading telemetry data into the network, either from the global Internet (external attacker) or from a compromised node within the Limited Domain (internal attacker). The second attack scenario is *telemetry interception*: an adversary collects sensitive telemetry data, either by exploiting insufficient boundary filtering (external attacker) or by intercepting traffic between trusted nodes via an on-path attack (internal attacker).

Both attack scenarios can be mitigated by encrypting and authenticating the IOAM telemetry data carried in the IPv6 Extension Header. Encryption prevents the disclosure of sensitive telemetry data in the event of interception, while authentication prevents the undetected insertion of misleading data. The following section presents our solution implementing these protections.

Proposed Solution

Since IOAM enables the collection of sensitive telemetry data, protecting the data gathered along the network path should be prioritized. We propose a solution based on an Authenticated Encryption with Associated Data (AEAD) scheme and circumvent the size limitation of a single IPv6 Hop-By-Hop option.

Encrypted and Authenticated IOAM Data

IOAM telemetry data should be both encrypted, to prevent disclosure in the event of interception, and authenticated, to ensure that an on-path adversary cannot inject misleading telemetry data into the protected network. We achieve both properties using an Authenticated Encryption with Associated Data (AEAD) [32] scheme, which natively provides both confidentiality and authentication. Specifically, our implementation supports both AES-GCM [24] and ChaCha20-Poly1305 [23, 27], both of which are available via the Linux kernel cryptographic API [19].

Each node within the IOAM Limited Domain uses one of the selected AEAD schemes, configurable from user space, to encrypt and authenticate its IOAM telemetry data with its own key, which is also configurable from user space. Each node then inserts its encrypted telemetry data, whose size is identical to that of the plaintext IOAM data, together with a 12-byte cryptographic nonce and a 16-byte authentication tag.

The AEAD computation performed by a node, for IOAM namespace α, can be formalized as follows:

$$(C, T) = \mathrm{AEAD}(K_\alpha, N_\alpha, A, P),$$

where:

- AEAD denotes the selected AEAD scheme (*i.e.*, AES-GCM or ChaCha20-Poly1305);
- K_α is the key associated with IOAM namespace α on the node;
- N_α is the incrementing nonce associated with IOAM namespace α on the node;
- A is the associated data;
- P is the plaintext IOAM telemetry data of the node;
- C and T are the resulting ciphertext and authentication tag, respectively.

Another approach would have been to use a single key shared by every node. While it simplifies key management for network administrators, it weakens the security of the proposed solution. In the event that the single shared key is leaked, the telemetry data of every node can be decrypted. Using a unique key per IOAM namespace and node prevents this potential issue and offers a more secure solution. The main drawback of this choice is that a potential telemetry collector must use the proper key for decrypting the encrypted telemetry data based on the plaintext IOAM namespace field of the header.

The proposed approach introduces two drawbacks. First, it requires additional per-packet computation for packets carrying secured IOAM telemetry data, due to the cryptographic operations required by AEAD. The resulting performance impact is evaluated in Sec. "Evaluation". Second, it requires additional space in the pre-allocated PTO, as shown in Fig. 6, to store the nonce and the authentication tag. This is problematic because a single IPv6 Hop-By-Hop option is subject to a size constraint, which we discuss in the following section.

Hop-By-Hop Option Size Constraint

The encapsulating node is responsible for adding an IPv6 Hop-By-Hop options Extension Header that carries the IOAM PTO option-type, as depicted in Fig. 6. The maximum size of a single Hop-By-Hop option is 257 bytes, including 2 bytes for the option type and length fields. Due to the alignment constraint, only 256 bytes of space remain in the header. After removing the 12 bytes required for the headers, 244 bytes are left available for IOAM node data.

Without any security mechanism, this space is sufficient for a reasonable number of nodes. However, with the proposed security solution, each node requires at least 32 bytes: a minimum of 4 bytes of encrypted IOAM data, a 12-byte nonce, and a 16-byte authentication tag. As a result, a single Hop-By-Hop option can accommodate at most 7 ($\lfloor 244/32 \rfloor$) nodes — a significant limitation in practice.

To circumvent this constraint, we multiply the size of the pre-allocated space by four compared to the value configured from user space, by modifying the IPv6 Extension Header size and reinterpreting the `RemainingLen` field in the IOAM PTO header as a multiple of sixteen bytes instead of four. This interpretation deviates from RFC 8200 [8], which specifies the standard semantics of this field; our current implementation is therefore a proof of concept whose goal is to evaluate the impact on forwarding performance, with a specification-compliant design left for future work. Under this modification, the available space for IOAM node data grows to 976 bytes ($244B \times 4$).

This extended space accommodates a significantly larger number of nodes, depending on the amount of IOAM data inserted per node:

- **Minimum data** (4 bytes per node): each node uses $4+12+16 = 32$ bytes, allowing up to 30 ($\lfloor 976/(4+12+16) \rfloor$) nodes.
- **Maximum data** (48 bytes per node, as in the current Linux implementation): each node uses $48+12+16 = 76$ bytes, rounded up to 80 bytes due to the 16-byte alignment imposed by the `RemainingLen` reinterpretation, allowing up to 12 ($\lfloor 976/80 \rfloor$) nodes.

Kernel-Space Implementation

One of the operation modes (*i.e.*, option-type) of IOAM, namely the Pre-allocated Trace Option-type (PTO), has already been implemented in the Linux kernel [18]. We extend this implementation with the proposed security solution while ensuring backward compatibility, and we release our implementation as open source. The implementation patch consists of a 700-line `git diff`, including context lines.

Gathering Plaintext IOAM Data

Before encrypting and authenticating the IOAM data pertaining to the node, it must first be gathered. Thus, we add a new function named `ioam6_gather_data` (see Listing 1). It is similar to the mainline `ioam6_fill_trace_data`, but it stores the plaintext IOAM data inside the given `buffer` instead of copying it directly into the packet. We patched the existing code to reuse as much of the existing implementation.

```
void ioam6_gather_data(struct sk_buff *skb, struct ioam6_namespace
     *ns, struct ioam6_trace_hdr *trace, bool is_input, u8
     *buffer);
```

Listing 1: New function for gathering plaintext IOAM data.

Encrypting and Authenticating IOAM Data

As presented in Sec. "Proposed Solution", the proposed security solution uses an AEAD scheme to encrypt and authenticate IOAM data. To do so, we added a new function, named `ioam6_sec_aead` (see Listing 2), in the kernel, which corresponds to $(C, T) = \text{AEAD}(K_\alpha, N_\alpha, \emptyset, P)$:

```
enum ioam6_sec_algo { AES_GCM, CHACHA };
int ioam6_sec_aead(enum ioam6_sec_algo algo, u8 *payload, u8
     *nonce, u8 *tag, struct net *net, struct ioam6_namespace *ns,
     struct ioam6_trace_hdr *trace);
```

Listing 2: New code for protecting IOAM data.

The enumeration `ioam6_sec_algo` in Listing 2 represents the AEAD schemes currently supported by the implementation of our proposed security solution. `ioam6_sec_aead` relies on the AEAD implementation in the cryptography API of the Linux kernel [19]. First, it obtains the AEAD transform stored in the internal data structure representing the IOAM namespace using the `ioam6_aead` function, described in Listing 4. Then, it creates an AEAD operation request by calling `aead_request_alloc` with the retrieved transform. The input of the AEAD operation is composed of the plaintext (`u8 *payload`) passed as a parameter to the function. Our implementation is not using Associated Data (AD). The output is composed of the ciphertext and the tag (`u8 *tag`). After the AEAD computation, the buffer pointed to by `payload` will contain the ciphertext, which has the same length as the plaintext. Afterwards, the AEAD computation is configured with the prepared input and output along with the given `nonce`. Finally, the AEAD computation is performed by calling `crypto_aead_encrypt` with the prepared request, which is freed with `aead_request_free` before exiting the function.

```
struct ioam6_namespace {
    struct rhash_head head;
    struct rcu_head rcu;

    struct ioam6_schema __rcu *schema;

    __be16 id;
    __be32 data;
    __be64 data_wide;

    atomic_t pkt_cnt;
    struct crypto_aead *aes;
    struct crypto_aead *chacha;
};
```

Listing 3: Structure representing an IOAM namespace. The last 3 fields are added by the proposed patch.

To reduce the impact on forwarding performance, the AEAD transforms for AES-GCM and ChaCha20-Poly1305 are initialized once per IOAM namespace — at namespace creation time (`ioam6_genl_addns`) — rather than being allocated and freed for each packet. The initialized transforms are stored directly in the namespace structure (`aes` and `chacha` fields in Listing 3) and reused for all subsequent packets, avoiding the per-packet overhead of `crypto_alloc_aead`. The authentication tag size is also configured at this stage.

A placeholder key consisting of all-zero bytes is set at initialization to satisfy the AEAD API requirements before a user-configured key is provided. This default key provides no security and must be replaced with a strong, randomly generated key before any deployment, as described in Sec. "Modifying AEAD Key".

```
struct crypto_aead *ioam6_aead(struct ioam6_namespace *ns, enum
     ioam6_sec_algo algo);
```

Listing 4: Function to get the required AEAD transform.

The helper function `ioam6_aead` (see Listing 4) returns the instance of the AEAD transform from `struct ioam6_namespace` (see Listing 3) for the given IOAM namespace and AEAD scheme. As a result of our design, a key is always associated with a single IOAM namespace, for both AEAD schemes.

Encapsulation

The first nodes on the path (*i.e.*, encapsulating nodes) need to allocate sufficient memory for the IOAM packet headers and, optionally, add their protected IOAM telemetry data to it. For encapsulating the headers required by IOAM PTO, the current implementation relies on the lightweight tunnel feature of the Linux kernel. The tunnel is configured from user space using `iproute2`, as depicted in Listing 7. The instantiation of the tunnel in the kernel, which will be created once the route is configured from user space, uses the same code path as the original implementation. However, if the security approach is enabled using the new kernel parameter `ioam6_encap_security`, as described in Sec. "New Kernel Parameters", the size of the IPv6 Extension Header is mangled, as described in Sec. "Hop-By-Hop Option Size Constraint". The actual size of the IOAM PTO data is equal to 16 times the `RemainingLen` field (instead of 4 times), inside the IOAM PTO header, as depicted in Listing 5.

```
if (!net->ipv6.sysctl.ioam6_encap_security)
    len_aligned = ALIGN(trace->remlen * 4, 8);
else
    len_aligned = ALIGN(trace->remlen * 4 * 4, 8);
```

Listing 5: Size of IOAM PTO data (*i.e.*, `len_aligned`) depending on the usage of the proposed security solution.

This field reinterpretation of the IOAM packets is effective only when the protection solution is enabled. The other operations at the encapsulating node are the gathering of the plaintext IOAM data and its protection (*i.e.*, encryption and authentication), as presented in Sec. "Gathering Plaintext IOAM Data" and Sec. "Encrypting and Authenticating IOAM Data".

The nonce, used by AEAD, must not be reused with the same key (see Sec. "AEAD " in the "Background" for more details). To ensure uniqueness, we use the packet counter of the lightweight tunnel as a nonce. It is stored in the state of the tunnel (`struct ioam6_lwt`) and incremented for each packet entering the tunnel.

Transit

Subsequent nodes on the path (*i.e.*, transit nodes) need to append their IOAM telemetry data inside the pre-allocated IOAM PTO header if one of their configured namespaces matches the namespace field of the packet header. If the security mechanism is not enabled on the input interface, the original code path for plaintext IOAM data is followed. Otherwise, if the security approach is enabled using the new per-interface kernel parameter (see Sec. "New Kernel Parameters" for more details), the new code path is taken.

Similarly to the code for the encapsulating node, first, the remaining space in the IOAM PTO data is checked to ensure that enough space is available. Then, the plaintext IOAM data is gathered and protected using, respectively, the functions `ioam6_gather_data` and `ioam6_sec_aead` presented in Sections "Gathering Plaintext IOAM Data" and "Encrypting and Authenticating IOAM Data". The protected IOAM data can be inserted along with the nonce and authentication tag inside the header.

The main difference with the code for the encapsulating node is the storage of the nonce. For the transit node, it is stored inside the state of the IOAM namespace as an `atomic_t` (see `pkt_cnt` in Listing 3). It is incremented every time an in-transit packet matching the IOAM namespace is protected.

Finally, if the security approach has been used, the offset used by the loop parsing the IPv6 Hop-By-Hop options (*i.e.*, function `ip6_parse_tlv`) must be mangled because we have reinterpreted the field `RemainingLen`, as depicted in Fig. 6, as a multiple of 16 bytes instead of 4. The modification is the addition of the second `if` block in Listing 6. `optlen` is the size of the IPv6 Hop-By-Hop option containing the IOAM headers and data. The size of the IOAM data is the size of the Hop-By-Hop option minus 12, which corresponds to the headers before the actual data (see Fig. 6 for the packet structure). The size of the option is taken into account at the end of the existing loop. However, since we have multiplied the size of the option by 4 (*i.e.*, Listing 5), it needs to be taken into account 3 additional times by increasing the offset in the packet and reducing the remaining number of bytes to parse.

```
if (!ipv6_hop_ioam(skb, off))
    return false;

if(READ_ONCE(__in6_dev_get(skb->dev)->cnf.ioam6_sec_enabled)) {
    int ptoLen = (optlen-12);
    off+=3*ptoLen;
    len-=3*ptoLen;
}
```

Listing 6: Modification of the offset for the loop (`ip6_parse_tlv`) parsing the IPv6 Hop-By-Hop options.

Decapsulation

In this initial proof-of-concept implementation, the IOAM decapsulating node does not perform any specific operation. In a revised version, the decapsulating node should verify the authentication tag associated with each node's data. If the verification succeeds, it should decrypt the corresponding telemetry data using the appropriate key. Consequently, the decapsulating node would need to have access to the keys associated with all encapsulating and transit nodes within the domain. Alternatively, the decapsulating node could also report the entire IOAM header to an external telemetry collector or validator to offload computations.

One limitation of the current implementation is the absence of an explicit identifier indicating the originating node of each IOAM trace entry. As a result, the decapsulating node cannot determine which key should be used for authentication and decryption. One possible solution would be to introduce an additional per-trace header containing the node identifier in plaintext, although such an extension would require proper standardization. Another possible approach would be to use a shared key for all nodes within the same namespace, which would weaken the overall security.

User-Space Implementation

In order to use protected IOAM telemetry data, additional configuration is required compared with the plaintext IOAM

PTO implementation [18]. To facilitate deployment, we allow system administrators to configure the kernel-space implementation using a combination of kernel parameters and `iproute2`.

Default Configuration

By building on the original IOAM PTO implementation in the Linux kernel [18], our implementation reuses the existing parameters required for plaintext IOAM configuration.

First, each machine acting as an IOAM-aware router must be configured with an IOAM node ID. Then, for each network interface, IOAM must be enabled and the corresponding interface ID must be configured. These parameters can be set using the following kernel parameters:

- `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`.

Afterwards, an IPv6 route must be configured to add an IPv6 Hop-By-Hop options Extension Header containing the IOAM PTO, using the lightweight tunnel API in the Linux kernel. This can be done by interacting with the kernel through the routing netlink interface. A simpler approach is to use `iproute2` with the following command:

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

Listing 7: Configuring an IPv6 route with IOAM PTO.

Finally, a common IOAM namespace, which represents a logical subdivision of the network domain, must be configured on each network node using `iproute2`:

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

Listing 8: Setting an IOAM namespace.

If one of the namespaces configured on a node within the domain matches the namespace field in the header of a received packet, and if IOAM is enabled on the input interface, the node adds its IOAM data to the pre-allocated space.

New Kernel Parameters

The following new kernel parameters enable IOAM data protection and allow the AEAD scheme to be selected:

- `net.ipv6.ioam6_encap_security`;
- `net.ipv6.ioam6_encap_security_chacha`;
- `net.ipv6.conf.{iface}.ioam6_sec_enabled`;
- `net.ipv6.conf.{iface}.ioam6_sec_chacha`.

The first and third parameters determine whether the default plaintext approach or the protected IOAM data approach is used. The first parameter applies to the encapsulating nodes, whereas the third one applies to the transit nodes.

Currently, AES-GCM and ChaCha20-Poly1305 are the only supported AEAD schemes. The second and fourth parameters select the AEAD scheme for encapsulating and transit nodes, respectively: when set to one, ChaCha20-Poly1305 is selected; otherwise, AES-GCM is used.

Modifying AEAD Key

As specified in the relevant technical specifications [32, 25, 27, 9], each $<K_\alpha, N_\alpha>$ pair must be unique when using an AEAD scheme. Reusing such a pair may enable an attacker to forge a valid ciphertext and/or authentication tag. To mitigate this risk, system administrators can update the `KEY` associated with a given IOAM namespace `ID`. This functionality is supported through the following extension to `iproute2`, which relies on the updated `ioam6_genl` user API and is accessible through Generic Netlink:

```
$> ip ioam namespace set ID key KEY
```

Listing 9: Configuring AEAD key.

Evaluation

Methodology

The main goal of this initial implementation is to assess whether securing IOAM telemetry data using an AEAD scheme significantly degrades IPv6 traffic forwarding performance. To evaluate our implementation, we built a testbed consisting of two physical servers interconnected port-to-port to form a loop. The servers use Intel XL710 40Gb/s network interface cards, each equipped with two ports. Their specifications are summarized in Table 1. The traffic generator runs TREX [7], while the other server, *i.e.*, the Device Under Test (DUT), is configured to maximize network performance[2].

Metric	Traffic Generator	DUT
RAM	32GB DDR4	16GB DDR4
Kernel	Linux 5.10	Patched Linux 6.18
CPU	Intel Xeon E5-2630v3 (8c/16t - 2.4GHz/3.2GHz)	

Table 1: Specifications of the servers used in the evaluation.

In the following, the term "*baseline*" refers to the maximum number of packets per second (*pps*) that the DUT can forward on a single CPU core without packet loss. Based on our measurements using standard 1500-byte (*i.e.*, default MTU on Linux) IPv6 packets without IOAM, the DUT can forward up to $930,000$ packets per second without any loss. Further details on the evaluation of this value are provided in the appendices. We intentionally enforced the forwarding of every packet using a single receive queue mapped to a single CPU core, in order to obtain a stable comparison basis. Hence, we can accurately measure the proportional impact of the proposed security solution on IPv6 packet forwarding.

[2]We release our test scripts as open source; they include the configuration of both machines.

Results

To evaluate the impact of the proposed security solution on packet forwarding performance, we measured the number of packets per second that the DUT can forward under different scenarios. These scenarios vary according to the IOAM injection rate, defined as the percentage of packets carrying IOAM data protected by the proposed security approach. Each experiment was run for 30 seconds for each combination of parameters.

Encapsulating node We evaluate the impact on packet forwarding when the encapsulating node adds the IPv6 Hop-By-Hop option for the IOAM PTO with different sizes of pre-allocated IOAM data space. As depicted in Fig. 2, the size of the pre-allocated IOAM data space has a similar impact on performance when the injection rate remains at or below 1%. At a 1% injection rate, this corresponds to an 8.6% performance degradation relative to the baseline forwarding of 1500-byte IPv6 packets without IOAM. Differences in packet forwarding performance across IOAM data-space sizes start to appear from an injection rate of 5%. At a 100% injection rate, a significant difference can be observed in the number of packets per second forwarded by the DUT. For a pre-allocated data space of 32 bytes (*i.e.*, the minimum size), the DUT forwards approximately 793,000 pps, whereas for a space of 976 bytes (*i.e.*, the maximum size), it forwards only approximately 425,000 pps. This corresponds to a performance loss of approximately 46.4% between the two sizes.

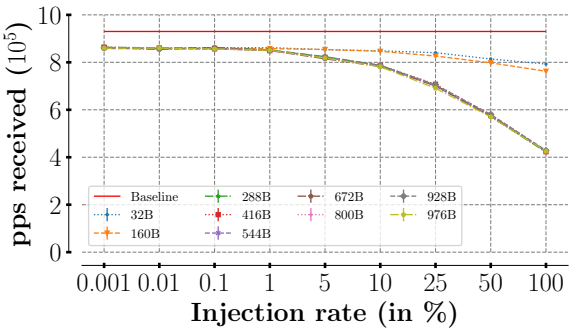

Figure 2: Encapsulation performance depending on the IOAM PTO size and injection rate.

Transit node For the transit node, we measure the number of packets per second that can be forwarded under two different scenarios. First, we study the impact of the selected AEAD scheme on packet forwarding performance, assuming that all IOAM data fields currently implemented in Linux (*i.e.*, 48 bytes of data) are protected. Second, we study the impact of the amount of protected IOAM data on packet forwarding performance, ranging from 4 bytes to 48 bytes.

As depicted in Fig. 3, the "No security" curve corresponds to the insertion of plaintext IOAM data (*i.e.*, the original implementation of IOAM) and achieves higher packet forwarding performance than the proposed security approach. This result is expected, as the security approach requires additional computations due to the AEAD scheme. For a 1% injection rate, the DUT forwards 930,000 pps with or without security. Enabling security protection incurs a negligible performance loss. For a 100% injection rate, the DUT forwards approximately 722,000 pps without security, whereas it forwards only between 435,000 and 600,000 pps when the security solution is used. This corresponds to a performance loss between 16.66% and 39.58% when security is enabled. However, a 100% injection rate is not expected to be used in a realistic deployment.

AES-GCM performs better than ChaCha20-Poly1305. This is an expected result since the CPU of the DUT (*i.e.*, an Intel Xeon E5-2630v3) has the AES hardware extension, namely AES-NI [16]. This extension provides dedicated CPU instructions for AES operations, allowing them to run on specialized hardware rather than as longer sequences of general-purpose instructions, resulting in faster computation.

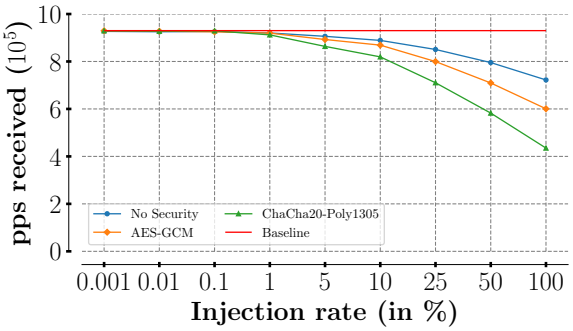

Figure 3: Transit performance depending on the chosen AEAD scheme and IOAM injection rate.

Secondly, we evaluate the impact of the number of protected IOAM data fields on IPv6 forwarding performance. As depicted in Fig. 4, the amount of protected data has a negligible impact. This result is expected, since the IOAM telemetry data size ranges between 4 and 48 bytes, corresponding to all IOAM data fields currently implemented in Linux.

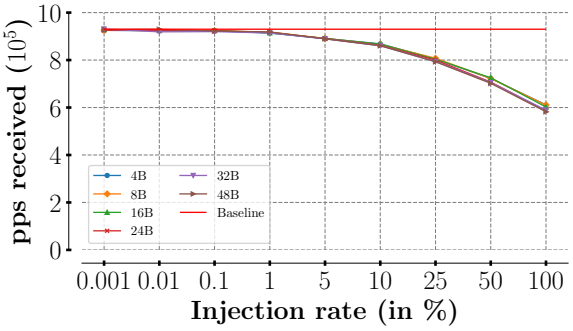

Figure 4: Transit performance depending on the number of IOAM data fields and injection rate using AES-GCM.

Profiling

After observing these results, we seek to determine which operations (*i.e.*, functions) are the most time-consuming for

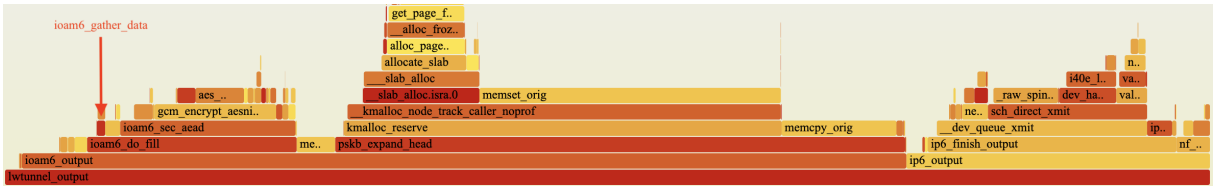

Figure 5: Flame graph of the functions executed in the patched Linux kernel when a packet enters the lightweight tunnel path.

the kernel. We run a similar experiment on the DUT acting as both an encapsulating and a transit node, with a PTO of the maximum size (*i.e.*, 976 bytes), all supported IOAM data fields (*i.e.*, 48 bytes of data to encrypt), and an injection rate of 100% (*i.e.*, IOAM is added to every packet). We profile the kernel on the DUT using the Linux `perf` tool and generate a `FlameGraph` [13] to identify the functions executed during packet processing and the execution time spent in each of them.

As shown in Fig. 5, the high-level function `ioam6_sec_aead`, which performs the AEAD computation on plaintext IOAM telemetry data, accounts for 9.75% of the execution time. The newly added `ioam6_gather_data`, which places the plaintext IOAM data in a stack buffer for the AEAD computation, and the existing `pskb_expand_head`, which inserts the required headers for the IOAM PTO, account for 0.47% and 31.68% of the execution time, respectively. Based on these results, we conclude that the AEAD computation has a significant impact on network traffic forwarding performance, which is expected, as cryptographic operations are known to be computationally intensive.

Related Work

Regarding the security of IOAM specifically, an active IETF Internet-Draft [4] proposes a solution to ensure the integrity of both the IOAM header and data for each standardized operation mode (*i.e.*, option-type), except IOAM DEX [34, 12]. Nevertheless, it does not address the confidentiality of IOAM data. Complementing this work, Iurman et al. [17] propose five additional integrity-focused solutions and implement three of them in the Linux kernel, including one that closely follows the Internet-Draft [4]. Each solution relies on AES-GMAC [24, 9], an integrity-only variant of AES-GCM, and their performance impact on IPv6 packet forwarding is evaluated using a methodology comparable to ours. However, none of these solutions address the confidentiality of IOAM data.

To the best of our knowledge, the present work is the first to provide both confidentiality and integrity protection for IOAM telemetry data, through a full AEAD scheme rather than an integrity-only construction.

Regarding other in-band telemetry protocols, among INT [20], AM-PM [10, 11], and ANT [28], only INT has been studied from a security perspective [21, 22, 30, 29, 31, 36, 38]. Several approaches have been proposed to secure INT data, but none of them provides a Linux kernel implementation, further distinguishing our work from the state of the art.

Conclusion

In Situ Operations, Administration, and Maintenance (IOAM) is a recently standardized network telemetry protocol that has yet to see widespread industry adoption. Although it carries sensitive network telemetry data, its security properties remain insufficiently studied. Given its early stage of deployment, this paper investigates how IOAM data can be protected. We propose, implement in the Linux kernel, and evaluate an approach for protecting IOAM data collected along the network path using an AEAD scheme based on AES-GCM or ChaCha20-Poly1305. Our evaluation shows that the impact on IPv6 traffic forwarding performance remains limited. In particular, for a realistic deployment scenario in which IOAM is used for 1% of in-transit packets, our proposed security solution incurs a negligible performance loss compared to the standardized plaintext IOAM.

In future work, we aim to refine the proposed solution into a version suitable for standardization within the IETF, *i.e.*, eliminating the modification of the IPv6 Hop-By-Hop option. We argue that standardizing a secure variant of IOAM is a crucial step toward enabling the safe deployment of the protocol across a broad range of industries.

Source Code

For reproducibility, the repository containing the source code for the kernel-space and user-space implementations described in this paper, together with the associated scripts, measurement data, and documentation, is publicly available at the following URL: `https://github.com/Advanced-Observability/ioam-security-kernel`

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

[2] Black, J. 2005. Authenticated Encryption. In van Tilborg, H., ed., *Encyclopedia of Cryptography and Security*. Springer. 11–21.

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

[4] Brockners, F.; Bhandari, S.; Mizrahi, T.; and Iurman, J. 2026. Integrity Protection of In Situ Operations, Administration, and Maintenance (IOAM) Data Fields. Internet-Draft (Work in Progress) draft-ietf-ippm-ioam-data-integrity-19, Internet Engineering Task Force.

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

[6] Carpenter, B. E., and Liu, B. 2020. Limited Domains and Internet Protocols. RFC 8799, Internet Engineering Task Force.

[7] Cisco. 2015. TRex. [Last Accessed: May 13th, 2026].

[8] Deering, S., and Hinden, B. 2017. Internet Protocol, Version 6 (IPv6) Specification. RFC 8200, Internet Engineering Task Force.

[9] Dworkin, M. 2007. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Technical Report NIST Special Publication (SP) 800-38D, National Institute of Standards and Technology.

[10] Fioccola, G.; Capello, A.; Cociglio, M.; Castaldelli, L.; Chen, M. G.; Zheng, L.; Mirsky, G.; and Mizrahi, T. 2018. Alternate-Marking Method for Passive and Hybrid Performance Monitoring. RFC 8321, Internet Engineering Task Force.

[11] Fioccola, G.; Cociglio, M.; Mirsky, G.; Mizrahi, T.; and Zhou, T. 2022. Alternate-Marking Method. RFC 9341, Internet Engineering Task Force.

[12] Goffart, M.; Wansart, E.; Iurman, J.; and Donnet, B. 2025. Linux Kernel Support for IOAM Direct Exporting. In *Proc. Technical Conference on Linux Networking (Netdev 0x19)*.

[13] Gregg, B. 2026. Flame Graph. [Last Accessed: May 8th, 2026].

[14] IANA. 2021. In Situ OAM (IOAM). [Last Accessed: May 7th, 2026].

[15] IANA. 2024. Internet Protocol Version 6 (IPv6) Parameters. [Last Accessed: May 7th, 2026].

[16] Intel. 2010. Securing the Enterprise with Intel AES-NI. White Paper 858748, Intel.

[17] Iurman, J., and Donnet, B. 2024. Implementing and Evaluating IOAM Integrity Protection. In *Proc. ACM/IRTF Applied Networking Research Workshop (ANRW)*.

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

[19] Kernel Development Community, T. 2026. Authenticated Encryption With Associated Data (AEAD). [Last Accessed: May 7th, 2026].

[20] Kim, C.; Sivaraman, A.; Katta, N.; Bas, A.; Dixit, A.; and Wobker, L. J. 2015. In-Band Network Telemetry via Programmable Dataplanes. In *Proc. ACM SIGCOMM Conference Posters and Demos*.

[21] Kong, D.; Zhou, Z.; Shen, Y.; Chen, X.; Cheng, Q.; Zhang, D.; and Wu, C. 2023. In-band Network Telemetry Manipulation Attacks and Countermeasures in Programmable Networks. In *Proc. IEEE/ACM International Symposium on Quality of Service (IWQoS)*.

[22] Kong, D.; Chen, X.; Lin, H.; Zhou, Z.; Shen, Y.; Liu, H.; Cheng, Q.; Liu, X.; Zhang, D.; Wu, C.; and Khan, M. K. 2026. Toward Security-Enhanced In-band Network Telemetry in Programmable Networks. *IEEE Transactions on Network and Service Management (TNSM)* 23(1):137–155.

[23] Langley, A.; Chang, W.-T.; Mavrogiannopoulos, N.; Strombergson, J.; and Josefsson, S. 2016. ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS). RFC 7905, Internet Engineering Task Force.

[24] McGrew, D. A., and Viega, J. 2005. The Galois/Counter Mode of Operation (GCM). Submission to NIST Modes of Operation Process. Revised submission, May 31, 2005.

[25] McGrew, D. 2008. An Interface and Algorithms for Authenticated Encryption. RFC 5116, Internet Engineering Task Force.

[26] Nir, Y., and Langley, A. 2015. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539, Internet Engineering Task Force.

[27] Nir, Y., and Langley, A. 2018. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, Internet Engineering Task Force.

[28] Pan, T.; Song, E.; Bian, Z.; Lin, X.; Peng, X.; Zhang, J.; Huang, T.; Liu, B.; and Liu, Y. 2019. INT-Path: Towards Optimal Path Planning for In-Band Network-Wide Telemetry. In *Proc. IEEE INFOCOM*.

[29] Pan, X.; Tang, S.; Liu, S.; Kong, J.; Zhang, X.; Hu, D.; Qi, J.; and Zhu, Z. 2020. Privacy-Preserving Multilayer In-Band Network Telemetry and Data Analytics: For Safety, Please Do Not Report Plaintext Data. *Journal of Lightwave Technology* 38(21):5855–5866.

[30] Pan, X.; Tang, S.; and Zhu, Z. 2020a. Multilayer Network Monitoring and Data Analytics over Encrypted Telemetry Data. In *Proc. IEEE International Conference on Communication Technology (ICCT)*.

[31] Pan, X.; Tang, S.; and Zhu, Z. 2020b. Privacy-Preserving Multilayer In-Band Network Telemetry and Data Analytics. In *Proc. IEEE/CIC International Conference on Communications in China (ICCC)*.

[32] Rogaway, P. 2002. Authenticated-Encryption with Associated-Data. In *Proc. ACM Conference on Computer and Communications Security (CCS)*.

[33] Salowey, J. A.; McGrew, D.; and Choudhury, A. 2008. AES Galois Counter Mode (GCM) Cipher Suites for TLS. RFC 5288, Internet Engineering Task Force.

[34] 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.

[35] Tan, L.; Su, W.; Zhang, W.; Lv, J.; Zhang, Z.; Miao, J.; Liu, X.; and Li, N. 2021. In-Band Network Telemetry: A Survey. *Computer Networks* 186:107763.

[36] Yang, K.; Chang, D.; Zhang, J.; and Niu, L. 2024. SecPro-INT: Secure Programmable In-Band Network Telemetry Method. In *Proc. International Conference on Computer and Communications (ICCC)*.

[37] Yu, M. 2019. Network Telemetry: Towards a Top-Down Approach. *ACM SIGCOMM Computer Communication Review* 49(1):11–17.

[38] Zhao, Y.; Cheng, G.; and Tang, Y. 2023. SINT: Toward a Blockchain-Based Secure In-Band Network Telemetry Architecture. *IEEE Transactions on Information Forensics and Security (TIFS)* 18:2667–2682.

Appendices

IOAM Packet Structure

IOAM can be encapsulated in various protocols, including IPV6 [1] or NSH [3]. Fig. 6 represents the packet structure of the IOAM PTO being used with IPV6 as a Hop-By-Hop options Extension Header [1].

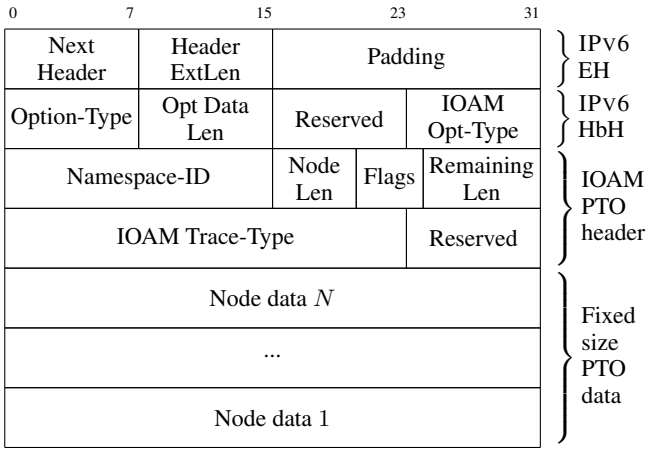

Figure 6: Single IPV6 Hop-By-Hop option containing the IOAM PTO [5, 1].

As discussed in Sec. "Kernel-Space Implementation", `HeaderExtLen` is mangled to circumvent the limitation on the size of a single IPV6 Hop-By-Hop option. Thus, allowing the PTO data space to reach up to 976 bytes. `Option-Type` is 49, which corresponds to IOAM as standardized by IANA [15]. We decided to use 0 as the `IOAM Opt-Type`, which is attributed to IOAM PTO [14], since we rely on the original IOAM PTO implementation in the Linux kernel [18]. `RemainingLen` has been modified and is now expressed in multiples of 16 bytes rather than 4 bytes.

IOAM Telemetry Data

The IOAM data fields, standardized in RFC 9197 [5], specify the telemetry information that may be collected at each IOAM node. The following list summarizes the IOAM data fields currently supported by the Linux kernel [18], as of version 7.0: `Hop_Limit`: Records the IPV6 Hop Limit value at the node; `Node_ID`: Identifies the node inserting the IOAM data. Available in 4 or 8 bytes (`Wide`) versions; `Ingress_Interface_ID` and `Egress_Interface_ID`: Identify the ingress and egress interfaces at the node. Available in 4 or 8 bytes (`Wide`) versions; `Timestamp Seconds` and `Timestamp Fraction`: Record the packet timestamp with second and sub-second precision; `Namespace-Specific Data`: Carries operator-defined data scoped to the IOAM namespace. Available in 4 or 8 bytes (`Wide`) versions; `Queue Depth`: Indicates the queue occupancy at the egress interface; `Opaque State Snapshot`: Allows nodes to export implementation-specific state information. This field has a variable length.

Additional IOAM data fields are standardized, but currently not supported in the Linux kernel: `Transit Delay`: Measures the time it takes to process the packet at the node; `Checksum Complement`: Used to maintain correctness of UDP checksums; `Buffer Occupancy`: Reports buffer utilization at the forwarding node.

Baseline IPV6 Forwarding

To evaluate the proportional impact of the proposed security solution on IPV6 forwarding, we evaluated the number of packets per second (*pps*) that the DUT (see Table 1 for the specifications) can forward before dropping any packet. We refer to that value as the "*baseline*". The packets are standard MTU-sized IPV6 packets without IOAM (*i.e.*, $1,500$ bytes on Linux). We used the same setup as in the other experiments, with a single 60-second run for each packet rate. To ensure a stable basis for comparison, we forced packet forwarding through a single receive queue bound to a dedicated CPU core. This was configured using the following command, where `in` denotes the receive interface:

```
$> ethtool -X in equal 1
```

Listing 10: Using a single receive queue.

First, to obtain an initial baseline value, we measured the packet drop rate when sending traffic between $100,000$ pps and 1 million pps, using increments of $100,000$ pps.

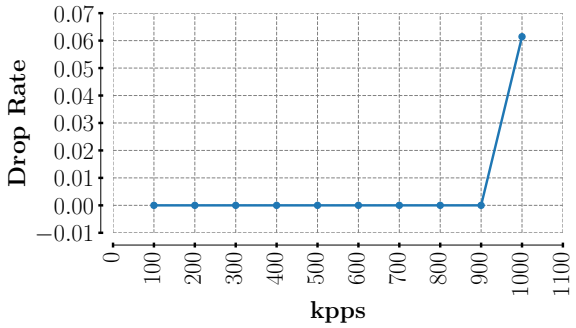

Figure 7: Broad evaluation of the baseline.

Based on this first experiment, as depicted in Fig. 7, we conclude that the baseline lies between $900,000$ pps and 1 million pps. We therefore ran a second experiment to obtain a more precise evaluation. Specifically, we evaluated the packet drop rate when sending traffic between $900,000$ and 1 million pps, using increments of $10,000$ pps.

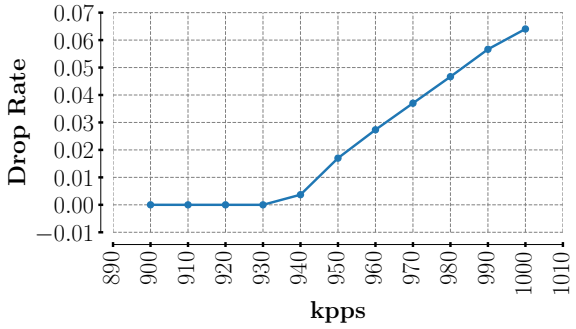

Figure 8: Precise evaluation of the baseline.

Based on Fig. 8, we conclude that the baseline lies between $930,000$ and $940,000$ pps. Therefore, in the other evaluation plots, we use $930,000$ packets per second as the baseline.