Implementing and Evaluating Ioam Integrity Protection

Justin Iurman
Université de Liège, Montefiore Institute
Belgium
justin.iurman@uliege.be

Benoit Donnet
Université de Liège, Montefiore Institute
Belgium
benoit.donnet@uliege.be

ABSTRACT

In-Situ Operations, Administration, and Maintenance (Ioam) gathers telemetry and operational information along a path, within packets. Up to now, Ioam header and data are carried in plain text without any protection against data-altering nodes or middleboxes. However, deploying Ioam in an untrusted or semi-trusted environment requires at least integrity protection. This paper leverages and analyzes work in progress about Ioam integrity protection and explains why the currently proposed solution can be improved. Accordingly, several alternative solutions are discussed, implemented in the Linux kernel, and evaluated. Based on the results, guidance is provided for standardization. Our source code is publicly available.

CCS CONCEPTS

• **Networks → Network measurement**; **Network manageability**; **Network monitoring**.

KEYWORDS

Ioam, Ipv6, integrity, validation, threats

ACM Reference Format:
Justin Iurman and Benoit Donnet. 2024. Implementing and Evaluating Ioam Integrity Protection. In *Applied Networking Research Workshop (ANRW 24), July 23, 2024, Vancouver, AA, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3673422.3674886

1 INTRODUCTION

These last years, *network telemetry* [25] has emerged as a mainstream technical term to refer to the newer network data collection and consumption techniques. Telemetry can be defined as an automated process for remotely collecting and processing network information. Network telemetry has been widely considered as an ideal mean to gain sufficient network visibility with better scalability, accuracy, coverage, and performance than traditional network measurement technologies.

While, historically, telemetry relied on explicit measurements [18] through, e.g., traditional BFD [14] or `traceroute`, a new paradigm has emerged, considering that intermediate hops can include telemetry data into regular packets, referring to *in-band telemetry* [23]. This has led to the definition of several in-band telemetry protocols, such as *In-band Network Telemetry* (Int) [15], *In-situ Operation Administration and Maintenance* (Ioam) [2], *Alternate Marking* [10], and *Active Network Telemetry* [19].

Those technologies require packet-level operations such as *encapsulation* (i.e., adding space for telemetry data), *data insertion* (i.e., inserting telemetry data), and *decapsulation* (i.e., removing the additional space for telemetry data). Such operations may threaten the confidentiality and security of user information. Kong et al. [16] have discussed threat models for Int. Firstly, Int can suffer from a control plane saturation attack caused from a compromised host. Secondly, nothing prevents attackers to launch a telemetry evading attack. Thirdly, attackers can leverage Int queue occupancy data collected by Int to conduct flooding attacks. Finally, person-in-the-middle attackers can sniff Int packets to collect data for preparing further attacks. In addition, Brockners et al. [4] performed a threat analysis for Ioam, including tampering of Ioam header and data, injection of fake Ioam header and data, and potential replay or delay attacks.

This paper focuses on the integrity validation of Ioam header and data. We leverage and analyze what is currently specified in an ongoing Ietf draft [4]. In particular, we provide the following contributions:

- while the Ietf draft only specifies a single solution, we introduce five additional potential ways to implement integrity validation for Ioam header and data, and explain why the current solution can be improved.
- we discuss the advantages and drawbacks of each solution, leading to a pre-selection of three candidates for implementation.

ANRW 24, July 23, 2024, Vancouver, AA, Canada
© 2024 ACM.
ACM ISBN 979-8-4007-0723-0/24/07
https://doi.org/10.1145/3673422.3674886

- we implement those three most promising solutions in the Linux kernel.[1]
- we evaluate those three solutions through extensive performance measurements, leading so to guidance for standardization.

The remainder of this paper is organized as follows: Sec. 2 provides the required background for this paper; Sec. 3 introduces several solutions to implement integrity validation for IOAM header and data; Sec. 4 discusses performance results of selected integrity solutions; Sec. 5 positions this paper with respect to the state of the art; finally, Sec. 6 concludes this paper by summarizing its main achievements.

2 BACKGROUND

Operations, Administration, and Maintenance (OAM) [18] refers to a set of techniques and mechanisms for performing fault detection and isolation, and for performance measurements. Throughout the years, multiple OAM tools have been developed for various layers in the protocol stack, going from basic `traceroute` to *Bidirectional Forwarding Detection* (BFD) [14]. Recently, OAM has been pushed further with *In-Situ OAM* (IOAM) [2]. The term "In-Situ" directly refers to the fact that the OAM and telemetry data are carried within packets rather than being sent through packets specifically dedicated to OAM. The IOAM traffic is embedded in data traffic, but not part of the packet payload.

In a nutshell, IOAM gathers telemetry and operational information along a path, within packets (see Fig. 2 where IOAM data is included between the IPV6 header and payload), as part of an existing (possibly additional) header. It is included in IPV6 packets as an IPV6 `Hop-by-Hop` extension header [5, 7]. 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 (node *A* in Fig. 2), remove (node *D* in Fig. 2), or update the extension header (nodes *B* and *C* in Fig. 2). IOAM data is added to a packet upon entering the domain and is removed from the packet when exiting the domain.

`RFC9197` [2] defines four IOAM *Option-Types*, for which different IOAM-Data-Fields are specified: (*i*) the *Pre-allocated Trace Option-Type*, where space for IOAM data is pre-allocated (see Fig. 2 where node *A* has prepared three "slots" for IOAM data), (*ii*) the *Incremental Trace Option-Type*, where nothing is pre-allocated and each node adds IOAM data while expanding the packet, (*iii*) the *Proof of Transit* (PoT) *Option-Type* to securely prove that the traffic passed through said defined path, and, (*iv*) the *Edge-to-Edge* (E2E) *Option-Type*. RFC9326 [22] defines the *Direct Export* (DEX) *Option-Type*, which is used as a trigger for IOAM data to be directly exported or locally aggregated without being pushed into in-flight data packets. Trace, PoT and DEX *Option-Types* are embedded in a `Hop-by-Hop` extension header, i.e., they are processed by every node on the path. On the contrary, the E2E *Option-Type* is embedded in a `Destination` extension header, i.e., it is only processed by the destination node. The aforementioned *Option-Types* are the only ones defined in the associated `IANA` registry [11] up to now.

[1]The source code is publicly available: https://github.com/iurmanj/ioam-integrity-linux-kernel

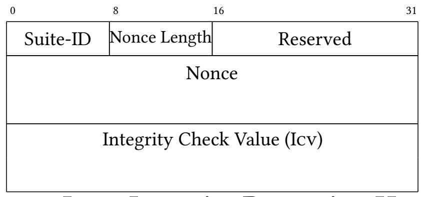


Figure 1: IOAM Integrity Protection Header.[2]

Currently, IOAM header and IOAM-Data-Fields (i.e., IOAM data), whatever the *Option-Type*, are carried in clear within packets, without protection against data-altering nodes or middleboxes. Deploying IOAM in an untrusted or semi-trusted environment requires at least integrity protection. Brockners et al. [4] have extended IOAM in order to carry IOAM header and data with integrity protection. Based on existing IOAM *Option-Types* listed in the `IANA` registry, they define new IOAM *Option-Types* (i.e., new code points in the registry) specifically for integrity protection.

For example, a new *Pre-allocated Trace Integrity Protected Option-Type* is defined and is the equivalent with integrity protection of the *Pre-allocated Trace Option-Type*. For these new *Integrity Protected Option-Types*, the equivalent *Option-Type* header is followed by a new header called *Integrity Protection Header*. In other words, the *Integrity Protection Header* sits between (*i*) the IOAM header and (*ii*) the IOAM data.

The *Integrity Protection Header* is illustrated in Fig. 1. The `Suite-ID` field defines the algorithm used for computing the integrity. The `Nonce Length` field provides the length of the nonce (provided in the `Nonce` field just below), in octets. Finally, the `Integrity Check Value` (abbreviated as ICV in the following) is the integrity value generated by the algorithm specified in the `Suite-ID` field.

3 INTEGRITY COMPUTATION AND VALIDATION

This section discusses different possibilities for computing and validating the integrity of the IOAM header and data. We compare those solutions (summarized in Table 1) and rely on Fig. 2 to explain each of them. Ultimately, we want a solution that works the same for all IOAM *Option-Types*.

[2]In this paper, both fields "Suite-ID" and "Integrity Check Value (ICV)" were renamed this way for better semantics.

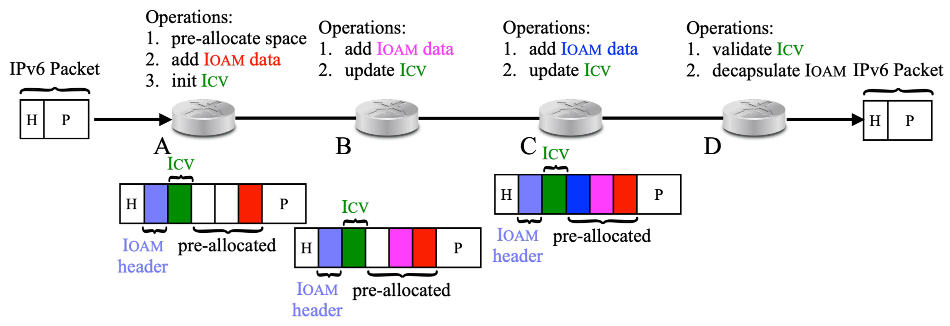


Figure 2: Generic example of IOAM data insertion and integrity computation. "H" corresponds to the IPV6 header, while "P" is the IPV6 payload. Node A pre-allocates the space (i.e., Pre-allocated Trace) for IOAM data, adds its own data, and initializes the ICV. Nodes B and C are transit nodes adding IOAM data in the pre-allocated space and updating the ICV. Finally, Node D performs the decapsulation (i.e., removes IOAM data) and validates the integrity.

Solution		Header		Full Protection		GMAC	
	# ICV	check?	freeze		# encap	# transit	# decap
1a	1	(✓)	✓	✗	1	p	n-1
1b	2	✓	✓	✗	2	2	n
1c	2	✓	✓	✗	1	2	n-1
2	1	(✗)	(✗)	✓	1	1	n-1
3	1	✓	✗	✗	1	2	1
4	1	✓	✗	✗	1	2	1

Table 1: Integrity validation comparison. Solutions 1.x corresponds to a validation at the end (with header check), solution 2 to a validation at the end (without header check), solution 3 to the neighbor validation, and solution 4 to IPSec. n refers to the number of nodes in the IOAM domain (from the encapsulation node to the decapsulation one; n=4 in Fig. 2), while p is the node position ($0 \leq p < n$) in the IOAM domain.

Therefore, we focus only on the integrity protection of the *Trace Option-Type*, due to its uniqueness. Other *Option-Types* work well with any solution presented in this section, which is not necessarily true for the *Trace Option-Type*.

Fig. 2 illustrates the three main actions associated to IOAM with integrity protection. Firstly, the *encapsulating* node (node A) pre-allocates IOAM space, adds data, and initializes the nonce and ICV. Secondly, the *transit* nodes (nodes B and C) add data and update the ICV. Thirdly, the *decapsulating* node (node D) validates the ICV and decapsulates IOAM data. How each node along the path handles the ICV is dependent on the way integrity is implemented. This is what we discuss in the next subsections.

The IETF draft authored by Brockners et al. [4] defines a solution (see Sec. 3.1, *Solution 1.a*) that uses a symmetric key based algorithm for integrity protection, i.e., the *Galois Message Authentication Code* [8] (GMAC). That decision has been made considering the space, performance, and operational constraints related to IOAM. Indeed, an asymmetric key based algorithm would not be suitable because of these constraints. GMAC is probably the best choice in this integrity protection context (e.g., it is used in IPsec [17]). Note that solutions we propose in this section also rely on GMAC, for the same reasons.

3.1 Solution 1: Validation at the End (Header Check)

In this solution, the integrity validation is performed at the end, by the *decapsulating* node, which is in charge of verifying the integrity of the IOAM header added by the *encapsulating* node, as well as IOAM-Data-Fields (i.e., IOAM data) added by each node on the path. Brockners et al. [4] define the following solution:

Solution 1.*a*: the encapsulating node (A on Fig. 2) picks a nonce and computes the initial ICV as a GMAC over the IOAM header and its IOAM data (i.e., ICV = GMAC (IOAM header || IOAM data)). Header fields that are mutable must be excluded from the GMAC. Next, each transit node along the IOAM path adds its own IOAM data and updates the ICV consequently, based on the previous ICV and its IOAM data (i.e., ICV = GMAC (ICV || IOAM data)). Finally, the decapsulating node checks the entire integrity chain by recomputing all intermediate ICVs. Thus, it requires as many computations as nodes present in the IOAM data.

This solution has serious drawbacks that could prevent its deployment. Table 1 highlights them, as for example: (*i*) in order to check the header, a transit node would have to compute p GMACs, with p being the number of previous IOAM nodes which added IOAM data (encapsulating node included); (*ii*) as a consequence, header check is made optional on transit nodes due to its complexity [4]. The only way to improve this solution would be to have two ICVs in the *Integrity Protection Header*, one for the IOAM header and one for the IOAM data, so that transit nodes could verify the integrity of the header in one step without recomputing the whole chain of ICVs up to themselves. This is what we propose with the following solution:

Solution 1.*b*: the encapsulating node picks a nonce and computes two GMACs, one for the IOAM header (HICV = GMAC (IOAM header)) and another for the IOAM data (DICV = GMAC (IOAM data)). Header fields that are mutable must be excluded from the GMAC. Next, each transit node along the IOAM path checks the integrity of the header based on HICV and, if successful, adds its own IOAM data and updates the DICV (DICV = GMAC (DICV || IOAM data)). Finally, the decapsulating node checks the entire integrity chain, i.e., both HICV and DICV.

As mentioned, the advantage of Solution 1.*b* over Solution 1.*a* is the ability for transit nodes to check the header integrity in a single shot. Consequently, it can be made mandatory instead of optional. However, as shown in Table 1, the encapsulating node has to perform two GMACs, which is not desirable. In order to improve it, we can simply modify the semantics of the two ICVs, i.e., one for the encapsulating node, and one for transit nodes. This is what we propose with the following solution:

Solution 1.*c*: the encapsulating node picks a nonce, initializes to 0 the ICV of transit nodes (TICV), and computes the ICV of the encapsulating node (EICV) as a GMAC over the IOAM header and its IOAM data (i.e., EICV = GMAC (IOAM header || IOAM data)). Header fields that are mutable must be excluded from the GMAC. Next, each transit node along the IOAM path checks the integrity of the header based on EICV and, if successful, adds its own IOAM data and updates the TICV (TICV = GMAC (TICV || IOAM data)). Finally, the decapsulating node checks the entire integrity chain, i.e., both EICV and TICV.

The advantage of Solution 1.*c* over Solution 1.*b* is that the encapsulating node only performs one GMAC, which is however still far from perfect. Indeed, for each transit node to check the integrity of the header, the IOAM data from the encapsulating node has to be fetched and included in the GMAC. Worse, this could seriously harm performance in some corner cases, e.g., when the *Opaque State Snapshot* is required in a *Trace Option-Type*.

Overall, those solutions bring too many trade-offs. As summarized in Table 1, Solutions 1.*b* and 1.*c* require two ICVs, leading to issues with IOAM space constraint. As for Solution 1.*a*, it is hardly deployable from a performance point of view. Also, none of them can be considered as "full" protection: it is not a zero trust scheme and, while non-IOAM nodes are not trusted, IOAM nodes are forced to be trusted (i.e., keys must be exchanged between IOAM nodes to validate the integrity of the header). This can be a problem if an IOAM node is compromised. Not to mention that these solutions "freeze" the header structures since they specify which header fields should be included or not in the GMAC, while future fields or flags would be kept out by default, which is an issue for interoperability. For all those reasons, our first advise would be that Solution 1.*a* and its derived should not be standardized.

3.2 Solution 2: Validation at the End (No Header Check)

In this solution, the integrity validation is still performed at the end, by the *decapsulating* node. However, there is no header check performed on transit nodes. Some header fields are still verified at the end by the *decapsulating* node. It works as follows:

Solution 2: the encapsulating node picks a nonce and computes the initial ICV as a GMAC over some selected IOAM header fields (e.g., *Namespace-ID*) and its IOAM data (i.e., ICV = GMAC (IOAM header-fields || IOAM data)). Next, each transit node along the IOAM path adds its own IOAM data and updates the ICV consequently, based on the previous ICV and its IOAM data (i.e., ICV = GMAC (ICV || IOAM data)). Finally, the decapsulating node checks the entire integrity chain by recomputing all intermediate ICVs.

As shown in Table 1, this solution addresses the drawbacks of Solutions 1.*x*. Here, transit nodes only perform one GMAC each, considering they no longer check the IOAM header. The trade-off of not checking the header on transit nodes actually makes a lot of sense: our primary objective is to provide integrity protection for IOAM data, not necessarily the header. Of course, some header fields are important and provide context to IOAM data, e.g., the *Namespace-ID* header field, which is common to all IOAM *Option-Types*. If altered, the IOAM data collected becomes meaningless. On the other hand, some header fields are only useful for processing. These two types of header fields should be distinguished, and only those that provide context to IOAM data should be protected by the encapsulating node. This also avoids any interoperability issues by no longer "freezing" header structures, since the selection of header fields is restrictive and sufficiently generic for current and future IOAM *Option-Types*, without any modification being required. Furthermore, this solution is the only one that corresponds to the desired "full" protection scheme, i.e., no node has to be trusted (IOAM or non-IOAM ones), except of course the decapsulating node which receives all the keys. Overall, this solution is considered a strong candidate and will be evaluated in Sec. 4.

3.3 Solution 3: Neighbor Validation

In this solution, the integrity validation is performed hop-by-hop on each IOAM node, which makes it easy to include the entire IOAM *Option-Type* (both the IOAM header and the IOAM data) in the GMAC. It works as follows:

Solution 3: the encapsulating node picks a nonce and computes the ICV as a GMAC over the entire IOAM *Option-Type* (i.e., ICV = GMAC (IOAM header || all IOAM data)). Next, each transit node along the IOAM path checks the integrity of the entire IOAM *Option-Type* based on the ICV and, if successful, adds its own IOAM data and updates the ICV consequently (i.e., ICV = GMAC (IOAM header || all IOAM data)). Finally, the decapsulating node checks the integrity of the entire IOAM *Option-Type* just like a transit node.

As shown in Table 1, the only drawback of this solution is that it does not correspond to the desired "full" protection scheme. Indeed, each IOAM node is responsible for verifying the integrity of the IOAM *Option-Type* received from its IOAM neighbor, which could be problematic in the case where an IOAM node is compromised. However, this solution also comes with a lot of advantages, such as: (*i*) the entire header can be checked, without the need to select certain fields; (*ii*) the validation load is shared between IOAM nodes instead of having a single node for validation; (*iii*) it is elegant, easy and quick to implement. Overall, this solution is considered a strong candidate as a lightweight alternative to Solution 2, and will be evaluated in Sec. 4.

3.4 Solution 4: IPSec

As shown in Table 1, this solution is quite similar to what Solution 3 proposes, except that it uses existing tools (i.e., IPSec [17]) without specifying new protocols. The drawbacks are the same as Solution 3, with the additional issue that IPSec tunnels must be configured between all nodes, not just between IOAM nodes. Even if this solution seems overkill compared to Solution 3, it is also considered a candidate and will be evaluated in Sec. 4.

4 EVALUATION

4.1 Methodology

In order to evaluate selected solutions' performance, we rely on TREX [6], an open source, low cost, stateful and stateless traffic generator fueled by DPDK. It has multiple advantages, such as the ability to generate Layer3–7 traffic and multiple streams, as well as the ability to easily craft your own packets with the underlying Scapy [21] layer. TREX can scale up to 200Gbps with only one server.

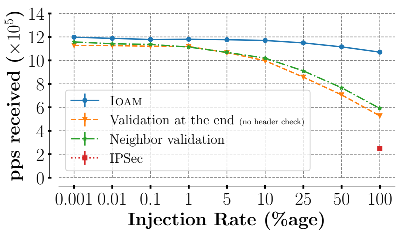
The testbed is straightforward: one machine for TREX, and another one for the *Device Under Test* (DUT). Both are equipped with an *Intel XL710 2x40GB QSFP+* NIC, each connected port to port on both ports in order to close the loop (i.e., TREX client and server run on the same machine). This kind of topology provides an easy way to evaluate IOAM and its different roles on the DUT, i.e., the encapsulating, transit, and decapsulating roles separately. The DUT has an *Intel Xeon CPU E5-2683 v4 at 2.10GHz*, with 16 Cores, 32 Threads, and has a 64GB RAM. During measurements, the DUT is configured to maximize its performance (e.g., cpu in performance mode, network settings). It is also configured to only use one queue for all traffic received, and so only one core responsible for that queue, in order to see the impact on a single core, which is better to compare performance on a common basis. Depending on the IOAM role and solution evaluated, the DUT is configured differently and TREX sends packets accordingly. Overall, each experiment (i.e., measurement) lasts 30 seconds and is run 20 times. We determine 95% confidence intervals for the mean based on the Student t distribution (they are too tight to be visible in the subsequent plots). Solutions 2, 3, and 4 (see Sec. 3) were implemented by leveraging the existing IOAM code [12, 13] in the Linux kernel.

4.2 Results

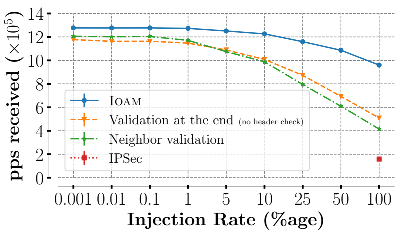
Fig. 3 shows results per IOAM role, respectively for an encapsulating node (Fig. 3a), a transit node (Fig. 3b), and a decapsulating node (Fig. 3c), and so for each solution we evaluated, i.e., Solution 2 ("*Validation at the end – no header check*", Sec. 3.2), Solution 3 ("*Neighbor validation*", Sec. 3.3), and Solution 4 ("*IPSec*", Sec. 3.4). On each graph, the X-axis represents the IOAM injection percentage and the Y-axis represents the number of received packets per second. The baseline represents vanilla IOAM, i.e., the current code in the Linux kernel without any integrity protection.

First, let us focus on the encapsulating role. Fig. 3a shows that Solution 2 ("*Validation at the end – no header check*") and Solution 3 ("*Neighbor validation*") both share similar performance, which is somehow expected since they perform one GMAC for this role (i.e., equal to the offset from the baseline). However, Solution 3 seems to perform a bit better, especially when IOAM is injected in more than 10% of the traffic. Such slight difference is explained by the simplicity of Solution 3 and the way it is implemented, compared to Solution 2 which is a bit more complex. Both solutions are suitable for line-rate when IOAM is added up to 1% of the traffic. After that, one notices increasing drops due to the GMAC. Regarding Solution 4 ("*IPSec*"), it is only represented for 100% (i.e., IOAM is added to all packets) due to the fact that an IPSec tunnel usually takes care of all the traffic without discrimination, even though some filters may be applied. Obviously, its performance is not good at all, but this is also the worst case (i.e., IOAM is applied to all packets). Overall, the encapsulating role seems to be the bottleneck for both Solutions 2 and 3.

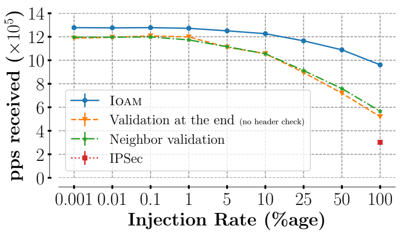
Then, let us focus on the transit role. Fig. 3b shows that Solution 2 ("*Validation at the end – no header check*") and Solution 3 ("*Neighbor validation*") both have, again, similar performance up to 10% of IOAM injection. After that, one clearly sees that Solution 3 becomes a bit less efficient, which is explained by the fact that Solution 2 performs one GMAC while Solution 3 performs two GMACs as a transit node. Both solutions are suitable for line-rate when IOAM is added up to 1% of the traffic. Without any surprise, Solution 4 ("*IPSec*") is terrible for 100% (i.e., which means one GMAC, one IPV6



(a) Encapsulating node (*A* on Fig. 2).



(b) Transit node (*B* or *C* on Fig. 2).



(c) Decapsulating node (*D* on Fig. 2).

Figure 3: Performance of IOAM integrity protection. The line labeled IOAM corresponds to vanilla IOAM, without any integrity protection, and serves as performance baseline for each role.

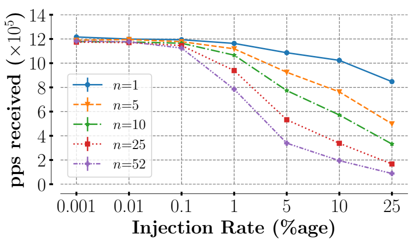


Figure 4: Validation at the end (no header check), zoom on the decapsulating node for a variable number of validations (i.e., GMACs).

decapsulation due to the IPSec tunnel mode, one IPv6 re-encapsulation, and one GMAC in that order). This role is therefore the bottleneck for Solution 4.

Finally, let us focus on the decapsulating role. Fig. 3c can be explained with the same reasoning as for the encapsulating role. However, the fact that Solution 2 ("*Validation at the end – no header check*") and Solution 3 ("*Neighbor validation*") both have similar performance may be surprising. Indeed, with Solution 3, only one GMAC is performed on the decapsulating node, while Solution 2 performs a variable number of validations (i.e., GMACs) depending on the number of nodes in the trace. In fact, as a common baseline in this figure, only one validation (i.e., GMAC) was performed by the decapsulating node for Solution 2, which corresponds to its best case scenario (i.e., similar to Solution 3). As a consequence, Fig. 4 also studies the impact of different number of validations (i.e., GMACs) for a decapsulating node with Solution 2. One interesting observation is that the maximum number of validations (i.e., 52) is suitable for line-rate up to 0.1%. Another interesting point is that five validations are suitable up to 1%. Based on discussions we had with operators, five nodes corresponds to the average IOAM domain length.

5 RELATED WORK

Up to now, only few works have been done to secure in-band telemetry technology. In particular, those works focus on the data plane programmability through INT [15]. Pan et al. [20] used vector homomorphic encryption to design a lightweight telemetry data encryption scheme. Wang et al. [24] rely on INT to identify an IP address spoofing. They, however, notice that the attacker may tamper with the measurement data, failing the detection. With SINT, Zhao et al. [26] propose a blockchain-based architecture to incorporate the security triad into the INT architecture. Finally, Kong et al. [16] discusses several attacks INT can suffer and propose Even-Mansour [9] block ciphering and SipHash [1] for integrity validation. With respect to IOAM, Brockners et al. [3] define mechanisms to securely prove that traffic transited said defined path. In this paper, we discussed multiple solutions for integrity validation in IOAM and implemented several of them into the Linux kernel for evaluating their performance.

6 CONCLUSION

In this paper, we analyze work in progress regarding the integrity protection of IOAM, and we discuss why the proposed solution can be improved. We propose five alternative solutions, select three of them as promising solutions for IOAM integrity protection, implement them in the Linux kernel, and evaluate them in a controlled environment.

We first confirm that Solution 2 ("*Validation at the end – no header check*") should be pushed towards standardization. However, the upcoming standard should not make mandatory for a decapsulating node to perform the validations and explain that validations can be delegated off-path to keep the forwarding path efficient. Regarding Solution 3 ("""*Neighbor validation*"""), it could be standardized later if an operator has a need, as a lightweight and intermediate solution. Finally, Solution 4 ("IPSec") could be discussed as a fallback solution.

ACKNOWLEDGMENTS

This work is supported by the CyberExcellence project funded by the Walloon Region, under number 2110186.

REFERENCES

[1] J.-P. Aumasson and D. J. Bernstein. 2012. SipHash: A Fast Short-Input PRF. In *Proc. International Conference on Cryptology in India (INFOCRYPT)*.

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

[3] F. Brockners, S. Bhandari, T. Mizrahi, S. Dara, and S. Youell. 2020. *Proof of Transit*. Internet Draft (Work in Progress) draft-ietf-sfc-proof-of-transit-08. Internet Engineering Task Force.

[4] F. Brockners, S. Bhandari, T. Mizrahi, and J. Iurman. 2024. *Integrity of In-Situ OAM Data Fields*. Internet Draft (Work in Progress) draft-ietf-ippm-ioam-data-integrity-08. Internet Engineering Task Force.

[5] B. Carpenter and S. Jiang. 2013. *Transmission and Processing of IPv6 Extension Headers*. RFC 7045. Internet Engineering Task Force.

[6] Cisco. [n. d.]. TRex: Realistic Traffic Generator. https://trex-tgn.cisco.com [Last Accessed: February 28th, 2024].

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

[8] M. Dworking. 2007. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. Special Publication 800-38D. National Institute of Standards and Technology.

[9] S. Even and Y. Mansour. 1997. A Construction of a Cipher from a Single Pseudo-Random Permutation. *Journal of Cryptology* 10, 3 (1997), 151–161.

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

[11] Internet Assigned Numbers Authority (IANA). [n. d.]. IOAM - Option-Type registry. https://www.iana.org/assignments/ioam/ioam.xhtml#option-type [Last Accessed: April 09th, 2024].

[12] J. Iurman. [n. d.]. IPv6 IOAM: Linux Kernel implementation. https://github.com/Advanced-Observability/ioam-linux-kernel [Last Accessed: June 7th, 2024].

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

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

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

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

[17] D. McGrew and J. Viega. 2006. *The Use of Galois Message Authentication Code (GMAC) in IPsec ESP and AH*. RFC 4543. Internet Engineering Task Force.

[18] T. Mizrahi, N. Sprecher, E. Bellagamba, and Y. Weingarten. 2014. *An Overview of Operations, Administration, and Maintenance (OAM) Tools*. RFC 7276. Internet Engineering Task Force.

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

[20] X. Pan, S. Tang, S. Liu, J. Long, X. Zhang, D. Hu, J. Qi, and Z. Zhu. 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 (November 2020), 5855–5866.

[21] Scapy Community. [n. d.]. Scapy. https://scapy.net [Last Accessed: April 20th, 2024].

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

[23] L. Tan, W. Su, W. Zhang, J. Lv, Z. Zhang, J. Miao, X. Liu, and N Li. 2021. In-Band Telemetry: A Survey. *Computer Networks* 186 (February 2021).

[24] R. Wang, Z. Wang, D. Wang, and Y. Liu. 2021. In-Band Network Telemetry Based Fine-Grained Traceability Against IP Address Spooling Attack. In *Proc. ACM Conference on Intelligent Computing and Its Emerging Applications (ICEA)*.

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

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