
Leveraging IPv6 Extension Headers for Advanced Network Observability

JUSTIN IURMAN



Department of Electrical Engineering and Computer Science
Faculty of Applied Science

UNIVERSITY OF LIÈGE - BELGIUM

SUPERVISOR: PROF. BENOIT DONNET

This dissertation is submitted for the degree of
DOCTOR IN COMPUTER SCIENCE

MAY 2026

Montefiore Institute

ABSTRACT

Modern networks are becoming increasingly complex, driven by high-speed links, virtualization, encrypted transport protocols, and the proliferation of globally distributed services. These networks rely on large-scale content distribution infrastructures, making their behavior highly dynamic and often difficult to predict. Operators must ensure both high availability and high-quality user experiences, yet traditional monitoring and measurement approaches struggle to provide timely, scalable, and fine-grained visibility. Active probing can be intrusive and limited in coverage, while passive collection often lacks sufficient detail to capture per-hop performance and path characteristics. This growing complexity and the limitations of conventional tools highlight the urgent need for advanced observability mechanisms that operate within the data plane, capture precise telemetry in real time, and remain deployable across production environments.

This thesis addresses these challenges by exploring the use of IPv6 Extension Headers and In Situ Operations, Administration, and Maintenance (IOAM) to enable robust observability. In IPv6, Extension Headers allow optional metadata to be carried in packets without modifying the core protocol, but their behavior in operational networks is inconsistent: routers and middleboxes may process or ignore Extension Headers, or worse, drop IPv6 packets containing Extension Headers. As for IOAM, which uses Extension Headers when used with IPv6, it allows packets to record per-hop operational data, capturing detailed latency, path, and network events in general. Beyond evaluating these mechanisms individually, this work investigates their combination to enable advanced observability, correlating network telemetry with application-level metrics for a more comprehensive, context-aware understanding of system performance. The main contributions of this thesis are: (i) a thorough evaluation of the survivability and processing behavior of IPv6 Extension Headers in controlled environments and at Internet scale; (ii) the implementation of IOAM in the Linux kernel, including performance evaluation; and (iii) the development and evaluation of a new technique that integrates network and application telemetry to provide richer operational insights. Together, these contributions advance the state of the art in observability, demonstrating how existing protocols and mechanisms can be leveraged to meet the evolving needs of modern networks while enabling operators to better understand, manage, and optimize complex infrastructures.

Keywords: IPv6; Extension Headers; Observability; IOAM;

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to Professor Benoit Donnet, my thesis supervisor, for giving me the opportunity to pursue a PhD and for his continuous guidance throughout these years. His invaluable advice, unwavering support, and availability have been key to the success of this work. I am also deeply grateful for his efforts in making conference and collaboration travels possible, always finding financial solutions when needed.

This research would not have been possible without the financial support of Cisco and the CyberExcellence project, to whom I express my sincere appreciation.

I am especially thankful to Frank Brockners for giving me the opportunity to work on IOAM and to become actively involved in the IETF. His mentorship and trust greatly shaped my professional path and led to a valued friendship that goes well beyond technical collaboration.

My sincere thanks also go to Eric Vyncke for his support and for offering me the chance to work on projects related to IPv6 Extension Headers. His experience within the IETF community has been a source of inspiration and learning. I am equally grateful for the friendship we have built over the years, from sharing a common passion for piloting airplanes to living just a few kilometers apart in Belgium.

I am grateful to my former colleagues, in particular Korian Edeline, Yves Vanaubel, and Cyril Soldani, for their mentorship when I joined the Research Unit in Networking (RUN) at the University of Liege. My thanks also go to Maxime Goffart and Emilien Wansart for their collaboration on IOAM-related work, which made the final stretch of this journey both smooth and enjoyable.

I would further like to thank all those who, directly or indirectly, contributed to the achievement of this thesis, including industry partners and collaborators from the IETF community. In particular, I wish to acknowledge Shwetha Bhandari, Tal Mizrahi, Thomas Graf, Benoit Claise, and Alex Huang Feng for their valuable discussions, technical input, and collaboration. I also greatly appreciate all other colleagues and contributors whose support and insights, even if not mentioned by name, have helped shape this work.

On a personal note, I owe immense gratitude to my family. To my wife, for her unfailing support, patience, and encouragement throughout this long endeavor. To my daughter, who was born as I reached the end of my PhD, and who has since been a constant source of joy and motivation. And to my parents, for their lifelong support and for instilling in me the curiosity and perseverance that made this possible.

From the bottom of my heart, thank you all!

AUTHOR'S DECLARATION

This thesis was written using the *University of Bristol Thesis Template* [26] created by V. BREÑA-MEDINA, and under the license *Creative Commons CC BY 4.0* [58]. The original template was modified in order to include parts and acronyms in the document.

TABLE OF CONTENTS

	Page
List of Tables	xiii
List of Figures	xv
Listings	xix
Acronyms	xxi
Introduction	i
Motivations	v
Main Contributions	v
Outline	x
I IPv6 Extension Headers	1
1 Background	3
1.1 IPv6 header	3
1.2 Extension Headers	4
1.2.1 Hop-by-Hop Options Header	7
1.2.2 Destination Options Header	10
1.2.3 Routing Header	10
1.2.4 Fragment Header	11
1.2.5 Authentication Header (AH)	12
1.2.6 Encapsulating Security Payload (ESP)	14
1.2.7 Others	15
1.3 Challenges related to Extension Headers	16
2 Related work	19
2.1 Standardization efforts	19
2.2 Studies and Measurements	20

TABLE OF CONTENTS

3	FISHNET	23
3.1	Implementation	24
3.2	Performance evaluation	27
3.2.1	Methodology	27
3.2.2	Results	28
3.3	Use case example	30
3.4	Possible improvements	31
4	Extension Headers Survivability in a controlled environment	33
4.1	Infrastructure Setup	33
4.2	Results	35
5	Extension Headers Survivability on the Internet	39
5.1	Measurement Methodology and Data Collection	39
5.2	Results	43
5.2.1	General Overview	43
5.2.2	Per Extension Header type	45
5.2.3	Drop Attribution	48
5.2.4	Impact on Path Length	51
5.2.5	Impact on Round-Trip Time	52
5.2.6	Discussion	53
6	Summary	55
II	Advanced Network Observability	59
7	Background	61
7.1	Telemetry	62
7.1.1	Application Telemetry	63
7.1.2	Network Telemetry	65
7.2	Monitoring vs. Observability	65
7.2.1	Monitoring tools	66
7.2.2	Observability tools	67
7.3	OAM Protocols and Tools	68
7.3.1	Active Measurement	68
7.3.2	Passive Measurement	69
7.3.3	Hybrid Measurement	70
7.4	Pushing further with Advanced Observability	70
8	In Situ Operations, Administration, and Maintenance (IOAM)	73

8.1	Terminology	73
8.1.1	IOAM-Domain	74
8.1.2	IOAM-Namespace	74
8.1.3	Encapsulating, Transit, and Decapsulating nodes	75
8.2	IOAM Option-Types	75
8.2.1	IOAM Pre-allocated Trace	75
8.2.2	IOAM Incremental Trace	77
8.2.3	IOAM Proof Of Transit (POT)	78
8.2.4	IOAM Edge-to-Edge (E2E)	79
8.2.5	IOAM Direct Export (DEX)	79
8.3	IOAM Deployment	80
8.3.1	IPv6 data plane	81
8.3.2	Challenges related to IPv6 tunnels	82
8.4	IOAM Use cases	84
8.4.1	Smart Traceroute: fault detection and isolation	85
8.4.2	M-Anycast: smart service selection and load-balancing for <i>microservices</i>	85
8.4.3	Others	87
8.5	Comparison with other hybrid measurements	88
8.5.1	IOAM vs. Alternate Marking	88
8.5.2	IOAM vs. INT	88
8.5.3	IOAM vs. HTS	89
9	IOAM implementations	91
9.1	Linux kernel	91
9.1.1	IPv6 packet processing and IOAM architecture	92
9.1.2	User space API (uAPI)	93
9.1.3	IOAM configuration	95
9.1.4	Performance evaluation	97
9.1.5	Improvements	100
9.2	Wireshark	101
9.3	Vector Packet Processor (VPP)	101
9.4	Cisco IOS	102
9.5	Huawei	102
9.6	Discussion	102
10	Bug discovery and fixing in the Linux kernel	103
10.1	Background	104
10.1.1	SRv6	104
10.1.2	RPL	105

TABLE OF CONTENTS

10.1.3	Socket Buffer	106
10.1.4	The <i>skb_cow_head()</i> function	107
10.2	Double-Reallocation Issue	107
10.3	Mitigation Solution	110
10.4	Performance evaluation	111
10.4.1	Methodology	111
10.4.2	Results	112
11	Integrity protection of IOAM	115
11.1	Threat model	116
11.2	Integrity Protection header	117
11.3	Integrity Protection solutions	117
11.3.1	Solution 1: Validation at the End (with header check on transit)	119
11.3.2	Solution 2: Validation at the End (no header check on transit)	122
11.3.3	Solution 3: Neighbor Validation	122
11.3.4	Solution 4: IPsec	123
11.4	Performance evaluation	123
11.4.1	Methodology	123
11.4.2	Results	125
11.5	Discussion	126
12	Cross-Layer Telemetry (CLT) for Advanced Observability	129
12.1	CLT Implementation	130
12.1.1	Version History	130
12.1.2	Kernel Space	131
12.1.3	User Space	132
12.1.4	Telemetry Data Collection and Processing	134
12.1.5	Architecture	134
12.2	Performance evaluation	135
12.2.1	Methodology	135
12.2.2	Results	136
12.3	Use case example	139
12.4	Discussion	142
13	Summary	145
	Conclusion	147
	Future Work and Research Directions	148
	Appendix A	151

TABLE OF CONTENTS

Appendix B	153
Appendix C	177
Appendix D	183
Bibliography	209

LIST OF TABLES

TABLE	Page
1.1 IANA’s Destination Options and Hop-by-Hop Options registry. [143]	9
1.2 IANA’s Routing Types registry. [143]	11
2.1 Comparison of related work with the contributions of this document. An empty-cell means no support, “~” means partial support, and ✓ means full support.	21
4.1 List of tested routers in our controlled environment.	33
4.2 Extension Headers survivability in the controlled lab, for each tested router. Each Extension Header is tested with different sizes when it makes sense. All tests are performed three times: one with UDP, one with TCP, and one with ICMPv6. Each ✓ corresponds to a successful test, <i>i.e</i> , when it successfully goes through the router and reaches the destination.	36
5.1 Extension Header types and sizes tested during a measurement campaign. Each × corresponds to an experiment, for a total of 38 experiments.	40
5.2 Measurement infrastructure running our experiments. 23 Cloud Providers (CPs) are considered for full mesh experiments, leading to 506 pair combinations, for a total of 403,788 traces per experiment (798 traces × 506 VP pairs). The “Label” column is used to easily identify each virtual machine (VM).	41
7.1 Examples of Metrics, Events, Logs, and Traces (MELT): Application vs. Network telemetry.	62
7.2 Difference in MELT coverage between monitoring and observability (network vs. application). An empty-cell means not supported, “~” means sometimes supported, and ✓ means supported.	65
7.3 Non-exhaustive list of well-known monitoring tools, with their MELT coverage and scope (network vs. application). An empty-cell means no support, “~” means limited support, and ✓ means full support. Based on native features only (no add-ons, plugins, or external modules).	66

LIST OF TABLES

7.4	Non-exhaustive list of well-known observability tools, with their MELT coverage and scope (network vs. application). An empty-cell means no support, and ✓ means full support. Based on native features only (no add-ons, plugins, or external modules). . .	67
8.1	IANA’s “IOAM Trace-Type” registry [142], which specifies the data types associated to each bit of the IOAM-Trace-Type field in Trace Option-Types (both Pre-allocated and Incremental).	77
8.2	IANA’s “IOAM E2E-Type” registry [142], which specifies the data types associated to each bit of the E2E-Type field in the E2E Option-Type.	79
8.3	IANA’s “IOAM DEX Extension-Flags” registry [142], which specifies the optional 4-octet fields associated to each bit of the Extension-Flags field in the DEX Option-Type. . .	81
8.4	Summary of IOAM options, depending on the IOAM Option-Type carried (<i>i.e.</i> , what type of Extension Header is used, and the different values of the “Option Type” and “IOAM Option-Type” fields).	82
10.1	Cases triggering a double-reallocation (<i>x86</i> architecture, Intel XL710 NIC).	109
11.1	Comparison of solutions for integrity protection of IOAM. Solutions 1.x correspond to a single validation at the end (with header check on transit), solution 2 corresponds to a single validation at the end (without header check on transit), solution 3 corresponds to the neighbor (<i>i.e.</i> , point to point) validation, and solution 4 corresponds to IPsec. An empty-cell means no support, ~ means partial support, and ✓ means full support. n refers to the number of nodes on a path in the IOAM domain (from the encapsulating node to the decapsulating node; $n=4$ in Fig. 11.3), while p is the node position ($0 \leq p < n$) in the IOAM domain.	118

LIST OF FIGURES

FIGURE	Page
1 Map of the Internet in 1982. [296]	ii
2 Visualization of the core IPv4 Internet topology in 2020. [35]	iii
3 Visualization of the core IPv6 Internet topology in 2020. [35]	iii
1.1 IPv6 Header. [64]	4
1.2 Example of a chain formed by the <i>Next Header</i> field in IPv6.	6
1.3 Hop-by-Hop Options Header. [64]	7
1.4 Option format. [64]	7
1.5 Destination Options Header. [64]	9
1.6 Routing Header. [64]	10
1.7 Fragment Header. [64]	12
1.8 Authentication Header (AH). [177]	13
1.9 Illustration of Transport and Tunnel modes for the Authentication Header (AH). . . .	14
1.10 Encapsulating Security Payload (ESP). [178]	14
1.11 Transport and Tunnel modes for Encapsulating Security Payload (ESP).	15
2.1 Timeline of IETF RFCs related to Extension Headers.	20
3.1 Overview of eBPF. [269]	24
3.2 Overview of how TC works with eBPF [283].	25
3.3 FISHNET commands (user program).	25
3.4 Testbed with TRES to evaluate FISHNET.	28
3.5 IPv6 Extension Headers injection with FISHNET and its impact on throughput. . . .	29
3.6 Impact on throughput between adding a <code>clsact</code> <code>qdisc</code> [197] and running a minimal version of a TC/eBPF program on egress.	29
4.1 Distribution of hardware vendors in the survey results.	34
5.1 Location of the virtual machines (VMs) running our experiments.	41

LIST OF FIGURES

5.2	Category of ASes traversed by our measurements (total is 64) and survey respondents. Important note: there is no intersection between the list of traversed ASes and the list of ASes that responded to the survey.	42
5.3	UpSet plot – IPv6 addresses collected during Step ₂ , Step ₃ , and Step ₄ of each Run, and the intersection between Runs.	42
5.4	High level overview of Extension Headers survivability between CPs, according to a specific upper-layer protocol. Results have been merged between the five Runs.	44
5.5	Survivability on a per Extension Header basis. Results have been merged over the five Runs.	46
5.6	Extension Headers drop attribution.	49
5.7	Example of uncertainty zones.	49
5.8	Path length comparison between Paris traceroute and Extension Headers (data from Step ₂ , Step ₃ , and Step ₄).	51
5.9	Round-trip time (RTT) comparison between Paris traceroute and Extension Headers (data from Step ₂ , Step ₃ , and Step ₄).	52
7.1	Overview of OpenTelemetry. [229]	63
7.2	Relation between traces and spans in distributed tracing.	64
8.1	Illustration of an IOAM-Domain.	74
8.2	IOAM Pre-allocated Trace example. “H” refers to the packet header, while “P” is the payload. Telemetry data (red and green) is inserted in the pre-allocated space. Router A is the <i>encapsulating</i> node, router B is a <i>transit</i> node, and router C is the <i>decapsulating</i> node.	75
8.3	IOAM Trace header (both Pre-allocated and Incremental).	76
8.4	IOAM Incremental Trace example. “H” refers to the packet header, while “P” is the payload. Telemetry data (red and green) is inserted in the trace. Router A is the <i>encapsulating</i> node, router B is a <i>transit</i> node, and router C is the <i>decapsulating</i> node.	77
8.5	IOAM Proof of Transit (POT) Type 0.	78
8.6	IOAM Edge-to-Edge (E2E) header.	79
8.7	IOAM DEX example. “H” refers to the packet header, while “P” is the payload. Router A is the <i>encapsulating</i> node, router B is a <i>transit</i> node, and router C is the <i>decapsulating</i> node.	80
8.8	IOAM Direct Export (DEX) header.	81
8.9	IOAM Option: IPv6 Hop-by-Hop and Destination Option format for carrying IOAM data.	81
8.10	Tunnel mode example for IOAM (<i>i.e.</i> , IPv6-in-IPv6 tunnel). “H” refers to the packet header, while “P” is the payload. The ingress (<i>i.e.</i> , the <i>encapsulating</i> node) encapsulates IOAM in an additional outer IPv6 header (<i>i.e.</i> , “OH”), and the egress (<i>i.e.</i> , <i>decapsulating</i> node) removes the outer header and forwards the inner IPv6 packet.	83

8.11	Inline mode example for IOAM. “H” refers to the packet header, while “P” is the payload. The ingress (<i>i.e.</i> , <i>encapsulating</i> node) directly inserts IOAM in IPv6 packets, and the egress (<i>i.e.</i> , <i>decapsulating</i> node) removes it.	84
8.12	Fault detection and isolation with IOAM.	86
8.13	M-Anycast: smart service selection and load balancing with IOAM and SRv6. [47] . .	86
8.14	Traffic SLAs and service/path verification with IOAM. [46]	87
8.15	Derive network traffic matrix with IOAM. [46]	87
9.1	The journey of an IPv6 packet through the Linux kernel network stack.	92
9.2	Testbed with TREN to evaluate IOAM performance in the Linux kernel.	97
9.3	Performance of the IOAM encapsulating role, depending on the encapsulation mode, the size of the pre-allocated trace, and the percentage of IOAM traffic.	98
9.4	Performance of the IOAM transit role, depending on the type of IOAM data field, and the percentage of IOAM traffic.	99
9.5	Performance of the IOAM decapsulating role, depending on the size of the pre-allocated trace, and the percentage of IOAM traffic.	100
9.6	Decoding of an IOAM Pre-allocated Trace Option-Type in Wireshark.	101
9.7	Decoding of an IOAM DEX Option-Type in Wireshark.	101
10.1	Segment Routing Header [91].	104
10.2	RPL Header [136].	105
10.3	Basic <i>sk_buff</i> diagram [180].	106
10.4	Testbed with TREN to evaluate the LWTs implementation of IOAM, SRv6, and RPL in the Linux kernel.	111
10.5	IOAM encapsulating role.	112
10.6	SRv6 encapsulating role.	113
10.7	RPL encapsulating role.	113
11.1	IOAM threat model: off-path vs. on-path attackers.	116
11.2	IOAM Integrity Protection header. ¹	117
11.3	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>i.e.</i> , Pre-allocated Trace) for IOAM data, adds its own data, and initializes the ICV. Nodes B and C are <i>transit</i> nodes adding IOAM data in the pre-allocated space and updating the ICV. Finally, Node D validates the integrity and performs the decapsulation (<i>i.e.</i> , removes IOAM).	118
11.4	Testbed with TREN to evaluate several integrity protection solutions for IOAM. . . .	123
11.5	Performance of IOAM integrity protection solutions. The line labeled IOAM corresponds to vanilla IOAM, without any integrity protection, and serves as performance baseline for each role.	124

LIST OF FIGURES

11.6	Solution 2: Validation at the end (no header check on transit) – zoom on the <i>decapsulating</i> node for a variable number of validations (<i>i.e.</i> , GMAC operations).	126
12.1	Call graph of <i>send</i> system calls for IPv6 and Layer-4 in the Linux kernel.	131
12.2	Enhanced IOAM Pre-allocated Trace header.	132
12.3	IOAM Pre-allocated Trace example. “H” refers to the packet header, while “P” is the payload. Telemetry data (red and green) is inserted in the pre-allocated space. Router A is the INGRESS of the IOAM domain, while C is the EGRESS.	133
12.4	Cross-Layer Telemetry (CLT) architecture.	135
12.5	Testbed with TREX to evaluate each component of CLT separately.	136
12.6	Time taken to enable or disable CLT with a Netlink call (Go vs. Python).	137
12.7	Performance of the IOAM agent.	137
12.8	Number of gRPC messages the IOAM collector is able to process, depending on their size.	139
12.9	Example of a simplified use case with CLT.	140
12.10	Example of an augmented span (with IOAM data) stored by Jaeger.	141
12.11	RTT measurement with and without CLT.	141

LISTINGS

LISTING	Page
3.1 Commands to attach an eBPF program to an interface on egress.	24
3.2 Simplified eBPF kernel program to inject a buffer of bytes in egress traffic.	26
9.1 IPv6 Option header for IOAM in the Linux kernel.	93
9.2 IOAM Pre-allocated Trace header in the Linux kernel (big-endian representation).	94
9.3 List of available <i>sysctls</i> to configure IOAM on a node.	95
9.4 Command to configure an IOAM-Namespace on a node.	95
9.5 Command to configure an IPv6 route with IOAM encapsulation.	95
9.6 Example of commands to configure the IOAM encapsulating role.	96
9.7 Example of commands to configure the IOAM transit role.	96
9.8 Commands to enable IPv6 tunnel decapsulation for the decapsulating role.	96
10.1 Definition of the <i>skb_cow_head()</i> function.	107
10.2 Simplified code pattern that may trigger two reallocations of socket buffers.	108
10.3 Simplified code pattern that mitigates the double-reallocation of socket buffers.	110
10.4 Code of the new <i>dst_dev_overhead()</i> function.	110
12.1 Simplified example of a Python application that uses OpenTelemetry for tracing, with additional annotations (lines 2 and 4) for CLT.	132
12.2 IOAM command-line configuration.	133
12.3 <i>tc</i> command to add a 150ms delay on an interface (e.g, eth1).	139
mq_qdepth.patch	177

ACRONYMS

AH	Authentication Header.
API	Application Programming Interface.
APM	Application Performance Management/Monitoring.
AS	Autonomous System.
ASIC	Application-Specific Integrated Circuit.
ASN	AS Number.
BCP	Best Current Practice.
BFD	Bidirectional Forwarding Detection.
BGP	Border Gateway Protocol.
BIER	Bit Index Explicit Replication.
BPF	Berkeley Packet Filter.
CDN	Content Delivery Network.
CNCF	Cloud Native Computing Foundation.
CP	Cloud Provider.
CPU	Central Processing Unit.
CRH	Compact Routing Header.
DCN	Data Center Network.
DEX	Direct Export.
DNS	Domain Name System.
DPI	Deep Packet Inspection.
DUT	Device Under Test.
E2E	Edge-to-Edge.
eBPF	extended BPF.
ECMP	Equal-Cost Multi-Path.
ECN	Explicit Congestion Notification.

ACRONYMS

EH	Extension Header.
ESP	Encapsulating Security Payload.
Geneve	Generic Network Virtualization Encapsulation.
GMAC	Galois Message Authentication Code.
GPE	Generic Protocol Extension.
GRE	Generic Routing Encapsulation.
GUI	Graphical User Interface.
HIP	Host Identity Protocol.
HTS	Hybrid Two-Step.
HTTP	HyperText Transfer Protocol.
HTTPS	HTTP Secure.
IANA	Internet Assigned Numbers Authority.
ICMP	Internet Control Message Protocol.
ICMPv6	ICMP for IPv6.
ICV	Integrity Check Value.
IETF	Internet Engineering Task Force.
INT	In-band Network Telemetry.
IOAM	In Situ OAM.
IOS	Internetworking Operating System (Cisco IOS).
IoT	Internet-of-Things.
IP	Internet Protocol.
IPFIX	IP Flow Information Export.
IPv4	Internet Protocol version 4.
IPv6	Internet Protocol version 6.
IS-IS	Intermediate System to Intermediate System.
ISP	Internet Service Provider.
IXP	Internet eXchange Point.
K8s	Kubernetes.
LISP	Locator/ID Separation Protocol.
LWT	Lightweight Tunnel.
MELT	Metrics, Events, Logs, and Traces.
MH	Mobility Header.

MPLS	MultiProtocol Label Switching.
MPTCP	Multipath TCP.
MQ	Multi-Queue.
MSS	Maximum Segment Size.
MTU	Maximum Transmission Unit.
NIC	Network Interface Card.
NPM	Network Performance Monitor.
NPU	Network Processing Unit.
NSH	Network Service Header.
OAM	Operations, Administration, and Maintenance.
OS	Operating System.
OSPF	Open Shortest Path First.
OTel	OpenTelemetry.
OTLP	OpenTelemetry Protocol.
OWAMP	One-Way Active Measurement Protocol.
P4	Programming Protocol-independent Packet Processors.
POT	Proof Of Transit.
QoE	Quality of Experience.
QoS	Quality of Service.
RFC	Request For Comments.
RH	Routing Header.
RIR	Regional Internet Registry.
RPL	Routing Protocol for Low-power and lossy networks.
RTT	Round-Trip Time.
SDK	Software Development Kit.
SFC	Service Function Chaining.
SLA	Service Level Agreement.
SR	Segment Routing.
SRH	Segment Routing Header.
SRv6	Segment Routing for IPv6.

ACRONYMS

SSS	Shamir's Secret Sharing.
STAMP	Simple Two-way Active Measurement Protocol.
STD	Standard.
TCP	Transmission Control Protocol.
TE	Traffic Engineering.
TG	Traffic Generator.
TLV	Type-Length-Value.
TWAMP	Two-Way Active Measurement Protocol.
UCMP	Unequal-Cost Multi-Path.
UDP	User Datagram Protocol.
UI	User Interface.
VM	Virtual Machine.
VP	Vantage Point.
VPN	Virtual Private Network.
VPP	Vector Packet Processor.
VXLAN	Virtual eXtensible Local Area Network.
VXLAN-GPE	GPE for VXLAN.
XDP	eXpress Data Path.
YANG	Yet Another Next Generation.

INTRODUCTION

The Internet Protocol defines the fundamental packet format and forwarding semantics used across the Internet. The first version, IPv4 [242], was designed with a variable-sized header that directly embeds optional information through IP Options [242]. As operational requirements evolved and the limitations of IPv4 became apparent, most notably its address space exhaustion [137], IPv6 [64] was standardized as the next-generation Internet Protocol. Beyond its vast address space, IPv6 comes with a 40-byte fixed-size header and introduces the mechanism of Extension Headers [64] (EHs). These Extension Headers make the protocol both extensible and adaptable to new requirements by allowing optional information to be carried in separate headers inserted between the IPv6 header and the upper-layer header, without altering the core protocol for more efficient processing. Since its inception, IPv6 deployment continues to increase worldwide [116].

In parallel, the Internet itself has undergone substantial transformation since the earliest studies on topology discovery [69]. At that time, Internet architecture was characterized by a hierarchical and sparsely interconnected network, exemplified by the design of the 1980s shown in Fig. 1. Over time, it has progressively evolved into a much flatter and more densely interconnected topology [66, 103, 304]. This transformation is clearly visible in Fig. 2 and Fig. 3, which show the core Internet topology for IPv4 and IPv6 respectively.

In this modern landscape, hypergiant distribution networks (*e.g.*, Akamai, Amazon, Cloudflare, Google, Meta, or Netflix), commonly referred to as hypergiants, have emerged as central actors. These large and globally distributed Content Delivery Networks (CDNs), operated by major technology companies, now account for a considerable share of all Internet traffic, driven by their constantly expanding user bases [25]. For example, Netflix reached 301 million [257] paid subscribers worldwide in 2024, which represents a 16% increase over the previous year. To sustain this demand while ensuring an optimal user experience (*i.e.*, low latency), hypergiants have deployed extensive and highly interconnected infrastructures spanning numerous data centers, relying heavily on Data Center Networks (DCNs) [43]. Such modern networks require accurate, timely, and scalable visibility into their operational state [226, 262]. As virtualization, encrypted transport protocols, and increasingly dynamic topologies become commonplace [16, 192], traditional monitoring approaches often fall short [226] of capturing the full range of network behaviors. Throughout the years, numerous Operations, Administration, and Maintenance (OAM) mechanisms have been introduced across different layers of the protocol stack [213]. These

POSTEL 25 FEB 82

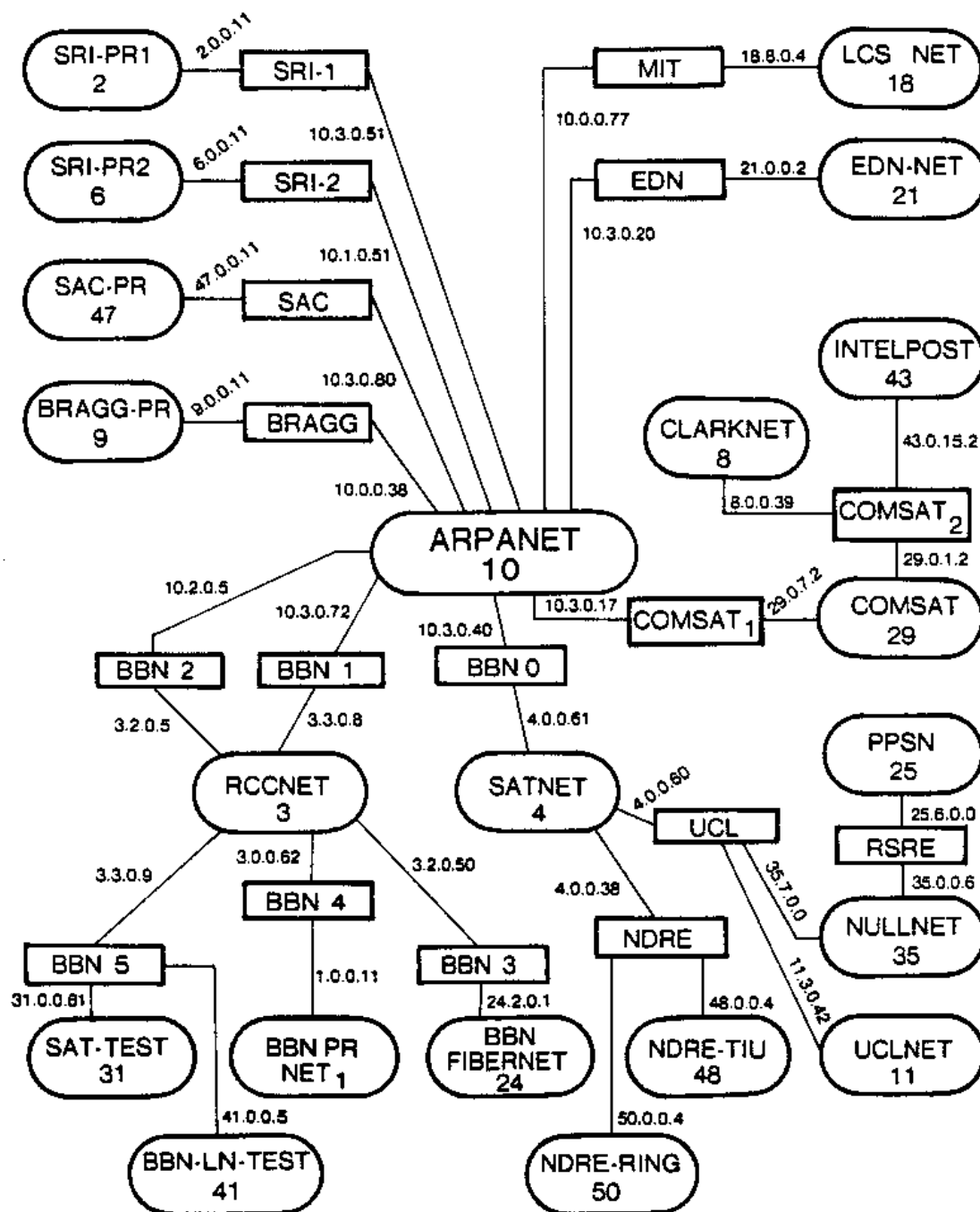


Figure 1: Map of the Internet in 1982. [296]

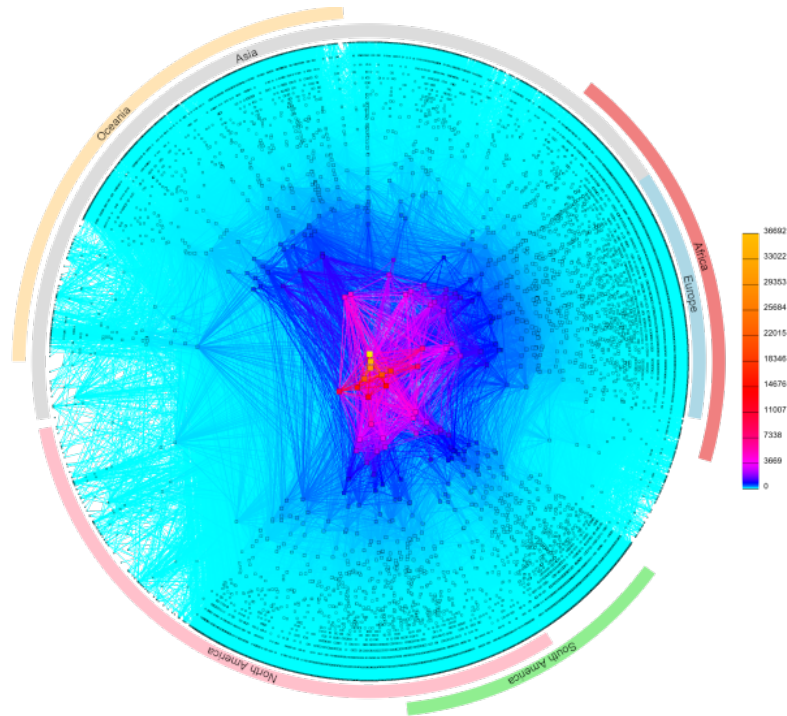


Figure 2: Visualization of the core IPv4 Internet topology in 2020. [35]

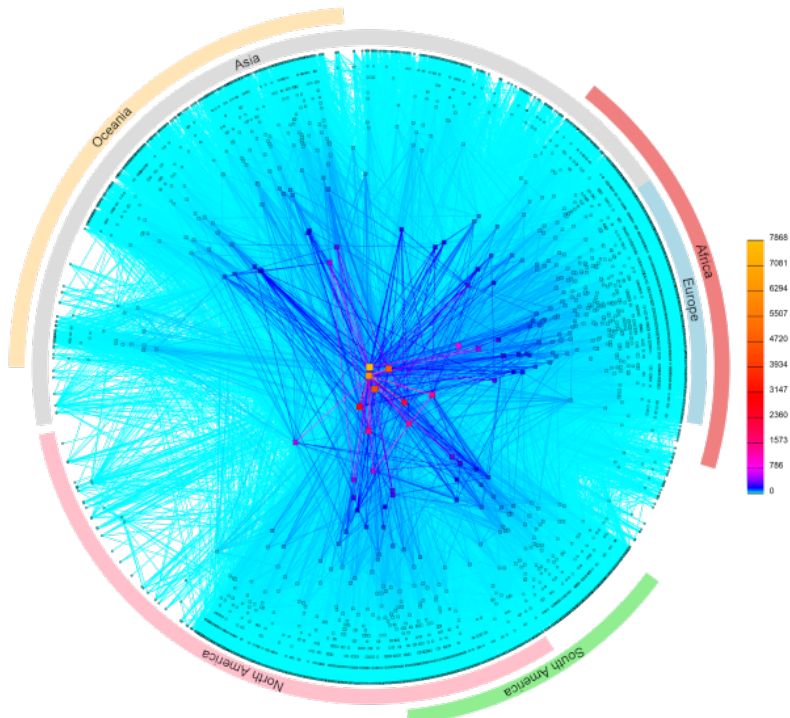


Figure 3: Visualization of the core IPv6 Internet topology in 2020. [35]

measurement techniques, developed under the OAM framework, enable fault detection and isolation, and provide precise performance measurements. Consequently, data collection (*i.e.*, *telemetry*, the technical term to describe the process of gathering and transmitting measured data from remote devices) is essential not only for hypergiants but also for DCNs and network operators in general, particularly to address two challenges [183, 302]. First, network infrastructures must maintain high availability, remaining operational even in the presence of failures. Second, they must deliver services with high quality and minimal latency. In order to meet these demands, operators must carefully design and operate their networks to maximize users' Quality-of-Experience (QoE). Together, these challenges make robust and scalable observability (*i.e.*, the ability of a system to make its internal state understandable through its telemetry data, which goes beyond monitoring) an indispensable requirement.

Concrete real-world events further illustrate these challenges. In late 2025, Netflix experienced brief but widely observed service disruptions [216] during the release of major *Stranger Things Season 5* episodes. Although the outages lasted only a few minutes, they coincided with highly synchronized access by millions of users and highlighted the difficulty of handling extreme, short-lived traffic surges in large-scale content delivery infrastructures. Such events demonstrate that even highly engineered and globally distributed CDNs remain vulnerable to transient overload and performance anomalies during peak demand, reinforcing the need for fine-grained, real-time network observability and adaptive load-balancing mechanisms.

In this context, the extensibility of IPv6 has enabled a variety of innovative technologies that leverage Extension Headers to implement new network functions. A prominent example is Segment Routing for IPv6 (SRv6) [91], which uses Extension Headers to encode explicit paths through the network. Technologically, SRv6 enables flexible source routing, traffic engineering, and service chaining without requiring additional per-flow state in the network core. Economically, it reduces operational complexity and cost by eliminating the need for complex signaling protocols, stateful middleboxes, and per-path management, allowing operators to deploy and scale services more efficiently. Another example is In Situ Operations, Administration, and Maintenance (IOAM) [18, 31], a new OAM mechanism recently standardized by the Internet Engineering Task Force (IETF) [148]. Appendix A provides more details on the IETF, an Internet standards body with thousands of published RFCs (*i.e.*, Request for Comments, which are technical specifications for the Internet), and its process. IOAM is an in-band measurement method that leverages IPv6 Extension Headers to embed telemetry (*i.e.*, operational data) inside packets as they traverse a network. This prominent measurement method allows collecting detailed telemetry about forwarding behavior, nodes, paths, and overall performance. As a result, IPv6 Extension Headers have emerged as a promising use case for observability, particularly in large-scale or performance-sensitive environments.

Motivations

A primary motivation of this thesis is to determine whether IPv6 Extension Headers can be reliably used. Although they provide a flexible mechanism for carrying metadata, their practical behavior on the Internet is inconsistent. Indeed, routers and middleboxes may process or ignore Extension Headers, or worse, drop IPv6 packets containing Extension Headers [59, 110, 124, 139, 140, 193, 290]. Understanding this variability, both in controlled environments and at Internet scale, is essential for assessing whether Extension Headers can support new technological capabilities, in particular for telemetry.

Another motivation is to evaluate the real-world feasibility of IOAM as an in-band telemetry protocol. IOAM enables packets to collect per-hop operational data, but its usefulness is determined by how feasible it is to implement, how straightforward it is to deploy, and how well it performs in real networks. Investigating whether IOAM can be efficiently integrated into widely used systems such as the Linux kernel, and whether its overhead remains acceptable in practice, drives a significant part of this research.

Finally, this thesis is motivated by the potential to push observability further by leveraging IOAM with IPv6 Extension Headers. Using Extension Headers to carry IOAM data enables in-band collection of fine-grained network telemetry, which can then be correlated with application-level metrics to provide a more comprehensive view of system performance. This combined approach provides richer, more precise, and context-aware insights than either network or application telemetry alone, allowing operators to better understand the interactions between the network and applications. Exploring this challenge and validating it through implementation and measurement forms a central direction of this work.

Main Contributions

This thesis aims to advance the state of the art in observability, namely advanced observability. To this end, the following Research Questions about IPv6 Extension Headers and IOAM must be answered:

RQ-1 *“Can we actually use IPv6 Extension Headers on the Internet?”*

RQ-2 *“What is the impact of IOAM on network traffic?”*

RQ-3 *“Can we ensure trust in IOAM data?”*

RQ-4 *“Can we achieve unified observability that correlates network and application layers?”*

The first Research Question (**RQ-1**) is answered in Chap. 6, at the end of the first part of this document, based on what is observed in Chap. 4 and Chap. 5. Our investigation shows that IPv6 Extension Headers can be used in controlled environments, but their reliability across

the wider Internet is limited. Measurements indicate that some routers systematically drop IPv6 packets containing Extension Headers based on the Extension Header type (*i.e.*, policies applied by operators) or size (*i.e.*, hardware limitation). While Extension Headers are technically supported in the IPv6 standard, practical deployment challenges limit their universal applicability. These findings suggest that Extension Headers can be used within controlled environments (*i.e.*, administrative domains, also referred to as limited domains) but require careful consideration and testing before Internet-wide deployment.

The second Research Question (**RQ-2**) is answered in Chap. 13, at the end of the second part of this document, based on what is observed in Chap. 9. Our experiments demonstrate that IOAM introduces a measurable impact on network traffic, particularly in software-based systems, due to the additional processing required to collect per-hop telemetry and attach metadata to packets. While this overhead can increase CPU utilization and slightly affect latency and throughput under high load, it remains within practical limits for most modern networks and can be mitigated through careful configuration and selective deployment. Our Linux kernel implementation and performance evaluations show that the cost of IOAM is predictable and manageable, making it a practical solution for in-band telemetry.

The third Research Question (**RQ-3**) is answered in Chap. 13, at the end of the second part of this document, based on what is observed in Chap. 11. Our work shows that IOAM data can be made trustworthy within a single administrative domain by adding integrity protection mechanisms that detect tampering or corruption in transit. We implement these mechanisms in the Linux kernel and evaluate their behavior. While integrity protection introduces additional processing and packet overhead, its impact remains manageable and can be mitigated through careful configuration and selective deployment. Our results demonstrate that the integrity protection provides reliable and verifiable telemetry without compromising the practicality of IOAM itself. Although end-to-end trust across multiple domains may require complementary techniques, this work confirms that operators can rely on trusted IOAM data in real deployments.

The fourth Research Question (**RQ-4**) is answered in Chap. 13, at the end of the second part of this document, based on what is observed in Chap. 12. Our work shows that, by leveraging IOAM and its use of IPv6 Extension Headers, it is possible to achieve unified observability, enabling detailed network telemetry to be correlated with application-level metrics. This integration allows operators to identify how network behavior affects application performance, revealing performance bottlenecks that would be difficult to detect otherwise. Through kernel-level implementation, testbed experiments, and application telemetry correlation, we demonstrate that this approach is feasible, efficient, and provides actionable insights, confirming that in-band telemetry can extend beyond the network layer to deliver a holistic view of system performance.

The publications written in the context of this research are listed below, in chronological order:

- *Implementation of IPv6 IOAM in Linux Kernel* [162]

- Iurman, J.** and Donnet, B. and Brockners, F.
Netdev Conference 0x14, July 2020
<https://hdl.handle.net/2268/239570>
- *IPv6 In-Situ Operations, Administration, and Maintenance* [158]
Iurman, J. and Donnet, B.
Software Impacts, November 2020
<https://hdl.handle.net/2268/252018>
 - *Towards cross-layer telemetry* [157]
Iurman, J. and Brockners, F. and Donnet, B.
ACM/IRTF Applied Networking Research Workshop (ANRW), July 2021
<https://hdl.handle.net/2268/261480>
 - *Measuring IPv6 Extension Headers Survivability with JAMES* [193]
Léas, R. and **Iurman, J.** and Vyncke, E. and Donnet, B.
ACM Internet Measurement Conference (IMC), Poster, October 2022
<https://hdl.handle.net/2268/294486>
 - *Cross-Layer Telemetry Support in Linux Kernel* [159]
Iurman, J. and Donnet, B.
Netdev Conference 0x16, October 2022
<https://hdl.handle.net/2268/296132>
 - *Using eBPF to inject IPv6 Extension Headers* [163]
Iurman, J. and Vyncke, E. and Donnet, B.
Netdev Conference 0x17, November 2023
<https://hdl.handle.net/2268/309796>
 - *Implementing and Evaluating IOAM Integrity Protection* [160]
Iurman, J. and Donnet, B.
ACM/IRTF Applied Networking Research Workshop (ANRW), July 2024
<https://hdl.handle.net/2268/319952>
 - *Linux Kernel Support for IOAM Direct Exporting* [105]
Goffart, M. and **Iurman, J.** and Wansart, E. and Donnet, B.
Netdev Conference 0x19, March 2025
<https://hdl.handle.net/2268/328634>
 - *Mitigating the Double-Reallocation Issue for IPv6 Lightweight Tunnel Encapsulations in the Linux Kernel* [164]
Iurman, J. and Wansart, E. and Goffart, M. and Donnet, B.

arXiv, March 2025

<https://arxiv.org/abs/2503.14959>

- *The Razor's Edge: IPv6 Extension Headers Survivability* [161]

Iurman, J. and Donnet, B.

Passive and Active Measurement (PAM) Conference, March 2025

Community Contribution Award

<https://hdl.handle.net/2268/325234>

Standardization efforts (*i.e.*, IETF documents) written in the context of this research are listed below, in chronological order (the order of the authors is irrelevant in this case):

- *In Situ OAM Profile for the Linux Kernel Implementation* [212]

Mizrahi, T. and **Iurman, J.** and Brockners, F.

Internet-Draft (individual), January 2022

<https://datatracker.ietf.org/doc/draft-mizrahi-ippm-ioam-linux-profile/>

- *Just Another Measurement of Extension header Survivability (JAMES)* [290]

Vyncke, E. and Leas, R. and **Iurman, J.**

Internet-Draft (individual), January 2023

<https://datatracker.ietf.org/doc/draft-vyncke-v6ops-james/>

- *Carrying an Identifier in IPv6 packets* [155]

Iurman, J.

Internet-Draft (individual), February 2023

<https://datatracker.ietf.org/doc/draft-iurman-6man-carry-identifier/>

- *RFC 9511: Attribution of Internet Probes* [289]

Vyncke, E. and Donnet, B. and **Iurman, J.**

RFC (Informational), November 2023

<https://datatracker.ietf.org/doc/rfc9511/>

- *Internet Control Message Protocol (ICMPv6) Loopback* [214]

Mizrahi, T. and Zhou, T. and Belkar, S. and Cohen, R. and **Iurman, J.**

Internet-Draft (individual), June 2024

<https://datatracker.ietf.org/doc/draft-mcb-6man-icmpv6-loopback/>

- *In Situ Operations, Administration, and Maintenance (IOAM) Template Option* [211]

Mizrahi, T. and Brockners, F. and Clemm, A. and **Iurman, J.** and Bhandari, S. and Zhou, T.

Internet-Draft (individual), July 2025

<https://datatracker.ietf.org/doc/draft-mbci-ippm-ioam-template-option/>

- *Integrity Protection of In Situ Operations, Administration, and Maintenance (IOAM) Data Fields* [34]
Brockners, F. and Bhandari, S. and Mizrahi, T. and **Iurman, J.**
Internet-Draft (ippm Working Group), May 2026
<https://datatracker.ietf.org/doc/draft-ietf-ippm-ioam-data-integrity/>
- *A YANG Data Model for In Situ Operations, Administration, and Maintenance (IOAM) Integrity Protected Options* [165]
Iurman, J. and Zhou, T.
Internet-Draft (ippm Working Group), May 2026
<https://datatracker.ietf.org/doc/draft-ietf-ippm-ioam-integrity-yang/>

The following contributions are also part of this research, not necessarily listed in chronological order:

- I prototyped FISHNET, a tool to inject Extension Headers in IPv6 traffic. [156]
<https://github.com/iurmanj/ebpf-ipv6-exthdr-injection>
- I participated to IETF standardization discussions and decisions about IPv6 Extension Headers. [84, 126, 161, 290]
- I implemented IOAM in the Linux kernel and added support to iproute2. Both were officially merged upstream. I also contributed to IPv6 in general. [171, 173]
<https://github.com/Advanced-Observability/ioam-linux-kernel>
- I added IOAM support to Wireshark. [172]
<https://github.com/Advanced-Observability/ioam-wireshark>
- I implemented the integrity protection of IOAM in the Linux kernel, as a proof-of-concept, and led its IETF standardization effort. [34, 170]
<https://github.com/Advanced-Observability/ioam-integrity-linux-kernel>
- I also led the IETF standardization of the YANG Data Model (see Appendix B) for the integrity protection of IOAM. [165]
- I designed Cross-Layer Telemetry (CLT) and implemented the entire IOAM ecosystem built around it (*i.e.*, API, agent, collector). [150–153]
<https://github.com/Advanced-Observability/cross-layer-telemetry>

Regarding long-term commitments, I am the official maintainer of IOAM in the Linux kernel, and I am still active within the IETF, mainly on matters related to IPv6, Extension Headers, OAM measurements, and IOAM specifically.

Outline

This document is divided into two parts. The first one covers IPv6 Extension Headers. More specifically, Chap. 1 provides the required technical background on IPv6, its Extension Headers mechanism and related challenges. Chap. 2 reviews the related work, while Chap. 3 introduces a new tool (*i.e.*, FISHNET) to inject Extension Headers in IPv6 traffic. Chap. 4 investigates whether IPv6 packets with Extension Headers pass through routers from different vendors with default configurations, while Chap. 5 performs large-scale measurements on the Internet to provide an overview of how IPv6 Extension Headers are processed and to compare the results with previous observations. Chap. 6 concludes the first part with a summary and answers the related Research Question.

The second part focuses on OAM measurements, more specifically on IOAM to improve observability (*i.e.*, advanced observability). Chap. 7 provides the required technical background to understand the difference between key terms such as telemetry, monitoring, and observability, and describes how they are achieved in practice. Chap. 8 presents IOAM in detail, covering terminology, its different options and how they work, deployment within IPv6 and related challenges, interesting use cases, and a comparison with similar protocols. Chap. 9 discusses existing IOAM implementations with particular focus on the Linux kernel and its performance evaluation, while Chap. 10 describes the analysis and fix of a bug related to IOAM in the Linux kernel. Chap. 11 explores and evaluates multiple solutions to ensure integrity protection of IOAM data, leading to an IETF contribution, while Chap. 12 demonstrates that it is possible, although complex, to correlate IOAM data (*i.e.*, network telemetry) with application telemetry. Chap. 13 concludes the second part with a summary and answers the related Research Questions.

Finally, the document ends with a conclusion that summarizes this research and discusses future work.

PART

I

IPv6 EXTENSION HEADERS

BACKGROUND

Over the past decade, IPv6 [64] has been more and more adopted [116]. While the latest version of the Internet Protocol was mainly designed to address the limitations of its predecessor IPv4, particularly the small address space leading to fast exhaustion [137], one of the critical innovations in IPv6 is its support for Extension Headers [64] (EHs). This mechanism allows the protocol to be both extensible and adaptable to emerging requirements, by providing a way to include optional information in separate headers that are placed between the IPv6 header and the upper-layer header in packets, therefore without modifying the core protocol (*i.e.*, the basic IPv6 header structure). Unlike IPv4 where Options [242] are included in the main variable-sized header, the IPv6 header remains a fixed size (*i.e.*, 40 bytes) for more efficient processing. IPv4 Options are listed in IANA's IP Option Numbers registry [141]. While most of them are deprecated [240], the remaining others generally do not work properly on the Internet, mainly for security reasons [106] (*i.e.*, packets with Options are often dropped [108]). Somehow, Options are to IPv4 what Extension Headers are to IPv6.

This chapter presents a comprehensive analysis of IPv6 Extension Headers, detailing their design, functionality, the specific roles for each type, and related challenges.

1.1 IPv6 header

Fig. 1.1 illustrates the IPv6 header format, as specified by RFC 8200, with the definition of the following fields:

- **Version** (4-bit) Internet Protocol version number (= 6).
- **Traffic Class** (8-bit) Used for traffic management, *e.g.*, Differentiated Services (DiffServ) [225] and Explicit Congestion Notification (ECN) [244].

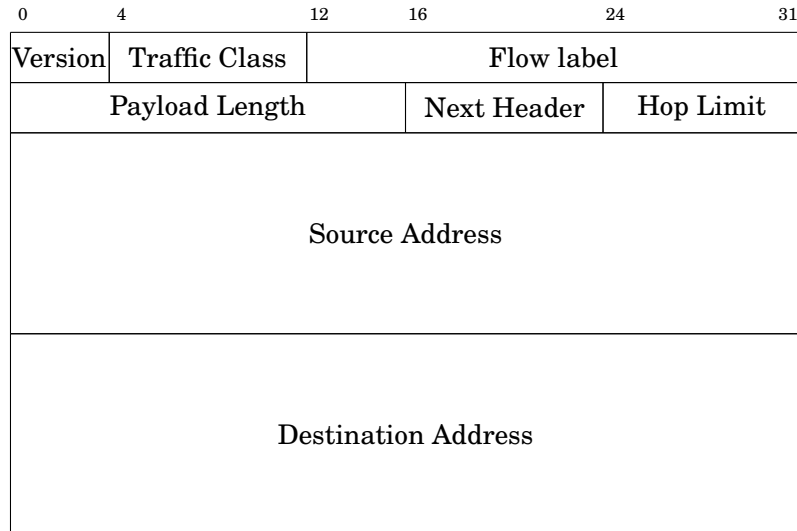


Figure 1.1: IPv6 Header. [64]

- **Flow Label (20-bit)** Used by a source to label sequences of packets to be treated in the network as a single flow. [8]
- **Payload Length (16-bit)** Length (in octets) of the IPv6 payload, *i.e.*, the rest of the packet following the IPv6 header. Any Extension Headers are considered part of the payload.
- **Next Header (8-bit)** Identifies the type of header immediately following the IPv6 header.
- **Hop Limit (8-bit)** Decrement by 1 by each node that forwards the packet. When forwarding, the packet is discarded if Hop Limit was zero when received or is decremented to zero.
- **Source Address (128-bit)** Address of the originator of the packet.
- **Destination Address (128-bit)** Address of the intended recipient of the packet.

Overall, the IPv6 header ensures proper routing of data across networks by carrying essential forwarding information that routers need to deliver packets to their destination. Extension Headers, on the other hand, represent optional information for additional per-packet functionality (*i.e.*, to extend IPv6).

1.2 Extension Headers

The IPv6 *Next Header* field identifies what type of header follows the IPv6 header, based on IANA's Protocol Numbers registry [145] (*e.g.*, 6 for TCP [72], 17 for UDP [241], 58 for ICMPv6 [55], or

even 59 for *No Next Header* [64] when there is no upper-layer header). Since all IPv6 EHs must share a uniform format [191], they always include a *Next Header* field as well with the same definition as the IPv6 header. This design allows for a chaining mechanism, *i.e.*, one or several Extension Headers between the IPv6 header and the upper-layer protocol. According to IANA's IPv6 Parameters registry [143], the following Extension Headers are currently defined, with their corresponding protocol number (*i.e.*, the value of the *Next Header* field):

- 0: Hop-by-Hop Options Header (Sec. 1.2.1)
- 43: Routing Header (Sec. 1.2.3)
- 44: Fragment Header (Sec. 1.2.4)
- 50: Encapsulating Security Payload (ESP) (Sec. 1.2.6)
- 51: Authentication Header (AH) (Sec. 1.2.5)
- 60: Destination Options Header (Sec. 1.2.2)
- 135: Mobility Header (Sec. 1.2.7)
- 139: Host Identity Protocol Header (Sec. 1.2.7)
- 140: Shim6 Protocol Header (Sec. 1.2.7)

They will be explained in more detail later on. Fig. 1.2 illustrates how it works in practice, based on three examples: the first one represents a TCP segment, the second one represents a Routing Header followed by a TCP segment, and the third one represents a Routing Header followed by an AH followed by a TCP segment.

At the time IPv6 was being designed in the IETF, one of the goals was to optimize the protocol for 64-bit processors. As a consequence, all Extension Headers must end on a 64-bit boundary (*i.e.*, 8-octet multiple), just like the IPv6 header. If an Extension Header is variable in length, it must use padding to ensure correct alignment. Multi-octet fields within each Extension Header are aligned on their natural boundaries.

The order of Extension Headers is also important. It is based on their definition and purpose (*i.e.*, how they are processed in the data plane), which makes their coexistence in packets possible and optimized for processing. RFC 8200 recommends the following order of appearance when multiple EHs are used in the same packet:

1. IPv6 header
2. Hop-by-Hop Options Header
3. Destination Options Header

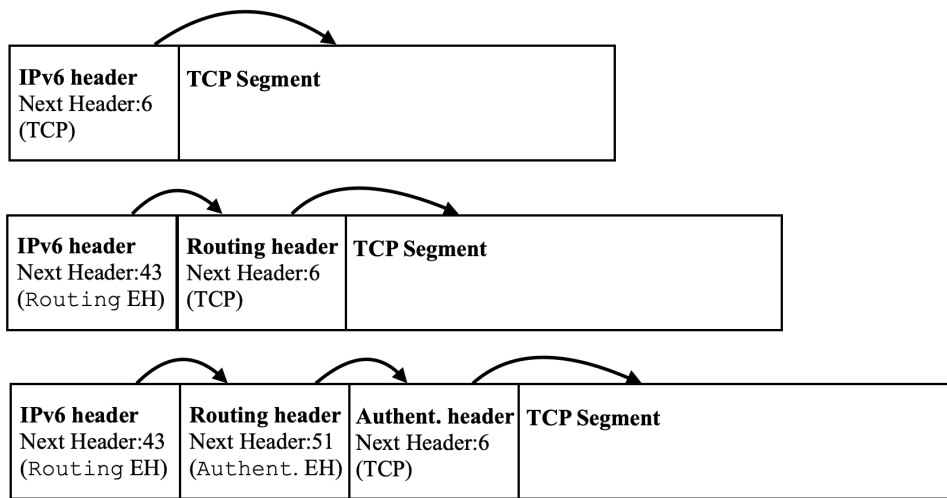


Figure 1.2: Example of a chain formed by the *Next Header* field in IPv6.

4. Routing Header
5. Fragment Header
6. Authentication Header (AH)
7. Encapsulating Security Payload (ESP)
8. Destination Options Header
9. Upper-layer header

Still among those recommendations, the Hop-by-Hop Options Header is restricted to only appear immediately after the IPv6 header. It appears first because it is the only Extension Header to be processed by every IPv6 node on the packet's path, unlike other Extension Headers which are only processed by the packet's destination. Also, each Extension Header should occur at most once, except for the Destination Options Header, which should occur at most twice (once in position 3 before a Routing Header, and once in position 8 before the upper-layer header). A Destination Options Header in position 3 is to be processed by the first destination that appears in the IPv6 *Destination Address* field, plus subsequent destinations listed in the Routing Header. A Destination Options Header in position 8 is to be processed only by the final destination of the packet. Finally, the Routing Header appears before the Fragment Header because the latter is only processed by the final destination, while the former is processed by intermediate destinations. As for the Authentication Header (AH) and Encapsulating Security Payload (ESP), they appear after the Fragment Header to ensure reassembly before expensive cryptographic operations. With this, security is applied once to the entire packet.

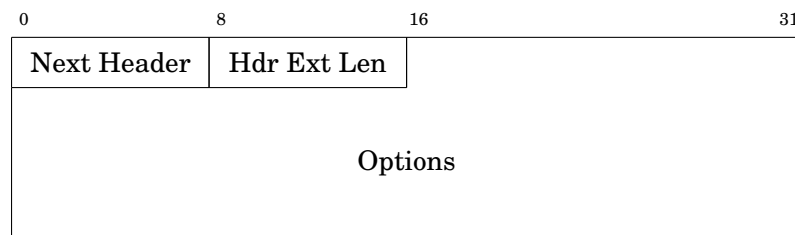


Figure 1.3: Hop-by-Hop Options Header. [64]

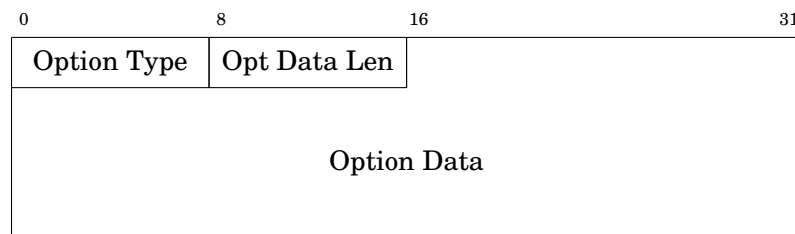


Figure 1.4: Option format. [64]

1.2.1 Hop-by-Hop Options Header

The Hop-by-Hop Options Header is used to carry optional information, called *Options*, that **may** [64] be examined and processed by every node along a packet's delivery path. It was previously mandatory in RFC 2460 (now obsoleted by RFC 8200), *i.e.*, *must* instead of *may*. The motivations behind such a change will be described over the next chapters.

Fig. 1.3 shows the header format of a Hop-by-Hop Options Header, as specified by RFC 8200, with the definition of the following fields:

- **Next Header** (8-bit) Identifies the type of header immediately following this header.
- **Hdr Ext Len** (8-bit) Length of this header in 8-octet units, not including the first 8 octets.
- **Options** Variable-length field that contains one or more TLV-encoded *Options*.

The minimum size of a Hop-by-Hop Options Header is 8 bytes, while the maximum size is 2048 bytes. Overall, the Hop-by-Hop Options Header is controversial because it places additional processing burden on every IPv6 node along a packet's path. A simple example is the Minimum Path MTU Hop-by-Hop Option [130] (Sec. 1.2.1.1), which is used to record the Minimum Path MTU along the path between a source and a destination. Each router along the path records the smallest MTU it can forward without fragmentation. The recorded value is then communicated back to the source.

1.2.1.1 Options

An option is type-length-value (TLV) encoded. RFC 8200 specifies that options must be processed in the order of appearance, and therefore cannot be skipped or ignored. However, it uses non-normative language for that, *i.e.*, “*must*” in lowercase letters, which may lead to different interpretations (*e.g.*, an implementation could skip or ignore options and remain RFC compliant). Fig. 1.4 shows the header format of an option, as specified by RFC 8200, with the definition of the following fields:

- **Option Type** (8-bit) Identifies the type of the option.
- **Opt Data Len** (8-bit) Length of the *Option Data* field of this option, in octets.
- **Option Data** Variable-length field that contains specific data depending on the *Option Type* field.

An option is identified by its full 8-bit Option Type. This field is encoded and can be broken down into three parts. The two highest-order bits specify the action to be taken when processing an unknown option: *00* to skip the option, *01* to discard the packet, *10* to discard the packet and send an ICMPv6 Parameter Problem (Code 2), or *11* to discard the packet and, if the Destination Address is not a multicast address, send an ICMPv6 Parameter Problem (Code 2). The third highest-order bit specifies whether the Option Data can be modified in transit: *0* if it does not change, or *1* if it may change. The remaining five low-order bits uniquely identify an option.

Table 1.1 shows the IANA’s Destination Options and Hop-by-Hop Options registry [143], which lists all the options currently defined. This document will not go into details regarding their format, purpose, etc, because it is off topic. Yet, some of them are covered by the second part of the document and will be discussed there. The registry includes some options that are deprecated, meaning their code points could be reclaimed (*i.e.*, redefined) in the future, in case of exhaustion (at the moment, there are only 12 unassigned 5-bit code points left). There are also some options considered as experiments (not to be confused with the RFC 3692-style Experiment [221] code point which is only used *for* experimentation and testing), meaning they are specified by RFCs with the experimental status, and some options that are very specific to some use cases and are, in some cases, not used anymore or pretty much dead. However, for the sake of completeness, the entire registry is listed. Unlisted code points are available for assignment.

The options *Pad1* and *PadN* are useful to meet alignment requirements, which are specific to each option. The former is used to insert a single octet of padding, while the latter is a TLV that is used to insert two or more octets of padding.

The minimum size of an option is 2 bytes with no data (or 1 byte for *Pad1*), while the maximum size is 257 bytes.

Table 1.1: IANA's Destination Options and Hop-by-Hop Options registry. [143]

Hex Value	Binary Value			Description	Reference
	act	chg	rest		
0x00	00	0	00000	Pad1	RFC 2460 [63]
0x01	00	0	00001	PadN	RFC 2460 [63]
0xC2	11	0	00010	Jumbo Payload	RFC 2675 [24]
0x23	00	1	00011	RPL Option	RFC 9008 [248]
0x63	01	1	00011	RPL Option (DEPRECATED)	RFC 9008 [248]
0x04	00	0	00100	Tunnel Encapsulation Limit	RFC 2473 [54]
0x05	00	0	00101	Router Alert (DEPRECATED)	RFC 9805 [21]
0x26	00	1	00110	Quick-Start	RFC 4782 [99]
0x07	00	0	00111	CALIPSO	RFC 5570 [264]
0x08	00	0	01000	SMF_DPD	RFC 6621 [199]
0xC9	11	0	01001	Home Address	RFC 6275 [237]
0x8A	10	0	01010	Endpoint Identification (DEPRECATED)	Expired [198]
0x8B	10	0	01011	ILNP Nonce	RFC 6744 [10]
0x8C	10	0	01100	Line-Identification Option	RFC 6788 [190]
0x4D	01	0	01101	Deprecated	RFC 7731 [135]
0x6D	01	1	01101	MPL Option	RFC 7731 [135]
0xEE	11	1	01110	IP_DFF	RFC 6971 [125]
0x0F	00	0	01111	Performance and Diagnostic Metrics (PDM)	RFC 8250 [77]
0x30	00	1	10000	Minimum Path MTU Hop-by-Hop Option	RFC 9268 [130]
0x11	00	0	10001	IOAM Option	RFC 9486 [18]
0x31	00	1	10001	IOAM Option	RFC 9486 [18]
0x12	00	0	10010	AltMark	RFC 9343 [98]
0x1E	00	0	11110	RFC3692-style Experiment	RFC 4727 [88]
0x3E	00	1	11110	RFC3692-style Experiment	RFC 4727 [88]
0x5E	01	0	11110	RFC3692-style Experiment	RFC 4727 [88]
0x7E	01	1	11110	RFC3692-style Experiment	RFC 4727 [88]
0x9E	10	0	11110	RFC3692-style Experiment	RFC 4727 [88]
0xBE	10	1	11110	RFC3692-style Experiment	RFC 4727 [88]
0xDE	11	0	11110	RFC3692-style Experiment	RFC 4727 [88]
0xFE	11	1	11110	RFC3692-style Experiment	RFC 4727 [88]

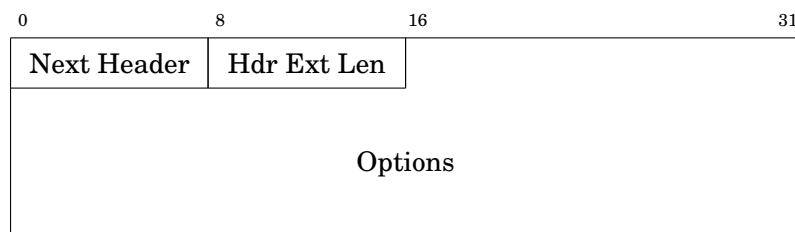


Figure 1.5: Destination Options Header. [64]

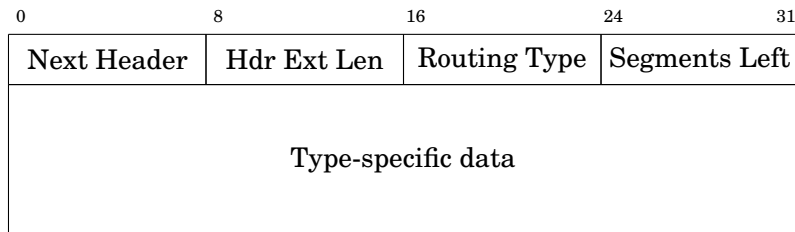


Figure 1.6: Routing Header. [64]

1.2.2 Destination Options Header

The Destination Options Header is used to carry optional information, called *Options*, that are examined only by a packet's destination node (*i.e.*, a node whose IPv6 address matches the IPv6 Destination Address field).

Fig. 1.5 shows the header format of a Destination Options Header, as specified by RFC 8200, which is exactly the same as for the Hop-by-Hop Options Header (see Sec. 1.2.1). The field definitions are also identical.

The minimum size of a Destination Options Header is 8 bytes, while the maximum size is 2048 bytes.

1.2.2.1 Options

Options (see Sec. 1.2.1.1) are listed in the IANA's Destination Options and Hop-by-Hop Options registry [143]. The same registry is used for the Hop-by-Hop Options Header and the Destination Options Header. In some cases, the specification of an option may restrict its use to only one of them.

1.2.3 Routing Header

The Routing Header allows the source of a packet to specify the path to the destination (*i.e.*, traffic steering), based on a list of one or more intermediate routers. It is the foundation for source routing mechanisms.

Fig. 1.6 shows the header format of the Routing Header, as specified by RFC 8200, with the definition of the following fields:

- **Next Header** (8-bit) Identifies the type of header immediately following this header.
- **Hdr Ext Len** (8-bit) Length of this header in 8-octet units, not including the first 8 octets.
- **Routing Type** (8-bit) Identifies a particular variant of the Routing Header.
- **Segments Left** (8-bit) Number of remaining segments to visit before the final destination.

Table 1.2: IANA’s Routing Types registry. [143]

Value	Description	Reference
0	Source Route (DEPRECATED)	RFC 5095 [1]
1	Nimrod (DEPRECATED)	Expired [245]
2	Type 2 Routing Header	RFC 6275 [237]
3	RPL Source Route Header	RFC 6554 [136]
4	Segment Routing Header (SRH)	RFC 8754 [91]
5	CRH-16	RFC 9631 [22]
6	CRH-32	RFC 9631 [22]
253	RFC3692-style Experiment	RFC 4727 [88]
254	RFC3692-style Experiment	RFC 4727 [88]
255	Reserved	–

- **Type-specific data** Variable-length field that contains specific format depending on the *Routing Type* field (and of length such that the complete Routing Header is an 8-octet multiple).

Table 1.2 shows the IANA’s Routing Types registry [143], which lists all the Routing Header types currently defined. Routing Header Type 0 (*Source Route*) was deprecated [1] due to security concerns (*i.e.*, amplification attacks and traffic loops), while Routing Header Type 1 (*Nimrod*) was never standardized and just expired [245]. Unlike other types which are of variable size, Routing Header Type 2 (*Mobile IPv6*) [237] is fixed size (*i.e.*, 24 bytes) as it allows for only one IPv6 address in the segments list. However, Mobility support in IPv6 never saw widespread adoption and is generally not used at all. The Compact Routing Header (CRH) [22] was recently standardized as experimental (both Routing Header Types 5 and 6). There are also two code points (*i.e.*, 253 and 254) for experimentation and testing, and a reserved code point (*i.e.*, 255). Unlisted code points are available for assignment (*i.e.*, from 7 to 252). Overall, few of them are widely deployed, with the possible exception of Routing Header Type 3 (RPL) [136] and, more particularly, Routing Header Type 4 (SRv6) [91]. These two are discussed in more details (*i.e.*, header format, purpose, etc) in the second part of this document.

The theoretical lower bound for the Routing Header size is 8 bytes, with zero segment. In practice, the minimum size of the Routing Header is 24 bytes (*i.e.*, one segment). The theoretical upper bound for the Routing Header size is 2048 bytes. In practice, the maximum size is 2040 bytes, with a maximum of 127 segments.

1.2.4 Fragment Header

Unlike IPv4, fragmentation in IPv6 is performed only by source nodes. Intermediate nodes (*e.g.*, routers) do not perform fragmentation. When a router receives an IPv6 packet larger than the MTU of its egress interface, it discards the packet and sends an ICMPv6 Packet Too Big error

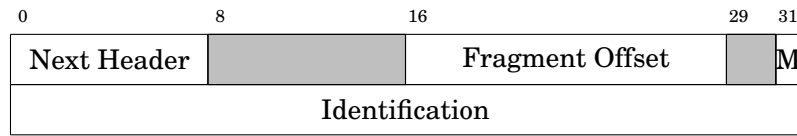


Figure 1.7: Fragment Header. [64]

message back to the source. The Fragment Header, which is fixed size (*i.e.*, 8 bytes), is used by an IPv6 source to send a packet larger than the path MTU (*i.e.*, the smallest MTU across a network path between the source and the destination). The destination of the packet reassembles the fragments.

Fig. 1.7 shows the header format of a Fragment Header, as specified by RFC 8200, with the definition of the following fields:

- **Next Header** (8-bit) Identifies the initial header type of the fragmentable part of the original packet.
- **Fragment Offset** (13-bit) The offset, in 8-octet units, of the data following this header, relative to the start of the fragmentable part of the original packet.
- **M flag** (1-bit) The M (More Fragments) flag indicates whether there are more fragments to follow (=1) or it is the last fragment (=0).
- **Identification** (32-bit) Identifies all fragments within the same original packet.

There are two categories of fragments: atomic and non-atomic. Atomic [107] fragments are IPv6 packets with a Fragment Header but without actual fragmentation (*i.e.*, Fragment Offset = 0, M flag = 0). Such packets are typically sent by sources that have received an ICMPv6 Packet Too Big error message that advertises a MTU smaller than 1280 bytes (*i.e.*, the IPv6 minimum link MTU specified by RFC 8200), which could happen for use cases with IPv6/IPv4 translators. Atomic fragments were deprecated [112] due to security concerns. Non-atomic fragments represent real fragmentation, *i.e.*, when a packet is divided into multiple pieces, each carrying a Fragment Header with proper Fragment Offset and M flag values. Despite being the standard for IPv6 fragmentation, even non-atomic fragments can be considered unsecured (*e.g.*, to bypass security devices). Many networks then drop IPv6 packets with a Fragment Header. Therefore, IPv6 fragmentation becomes almost impossible and a source would usually rely only on the signaled path MTU.

1.2.5 Authentication Header (AH)

The Authentication Header (AH), part of the IPsec [179] framework, is used to provide data origin authentication and integrity for IP datagrams, as well as protection against replays.

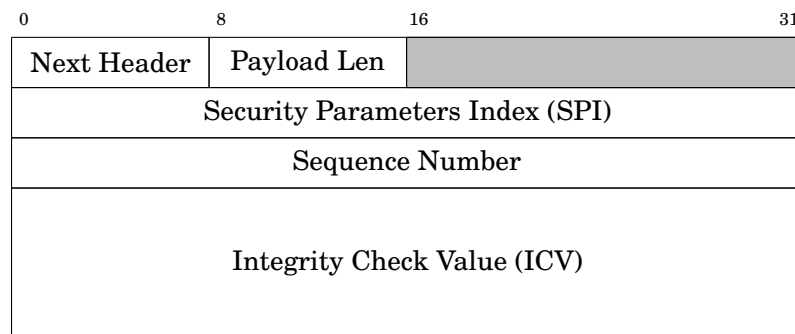


Figure 1.8: Authentication Header (AH). [177]

Fig. 1.8 shows the header format of an Authentication Header (AH), as specified by RFC 4302, with the definition of the following fields:

- **Next Header** (8-bit) Identifies the type of header immediately following this header.
- **Payload Len** (8-bit) Length of this header in 4-octet units, not including the first 8 octets.
- **SPI** (32-bit) Arbitrary value that is used by a receiver to identify the Security Association (SA) to which an incoming packet is bound.
- **Seq. Number** (32-bit) Contains a counter value (per SPI) that increases by one for each packet sent.
- **ICV** Variable-length field that contains the Integrity Check Value for the packet.

The theoretical lower bound for the Authentication Header (AH) size is 12 bytes, with no ICV (which makes no sense), or 16 bytes, with a minimum 4-byte ICV (which does not exist). In practice, it must be an 8-octet multiple and the minimum size of the Authentication Header (AH) is 24 bytes (*i.e.*, the smallest ICV is 12 bytes), although it depends on the algorithm. The theoretical upper bound for the Authentication Header (AH) size is 1028 bytes. In practice, the maximum size is 1024 bytes (*i.e.*, so that it is an 8-octet multiple, as required by IPv6).

1.2.5.1 Transport and Tunnel modes

The Authentication Header (AH) can work in two different modes: transport and tunnel modes. Fig. 1.9 illustrates the difference between these two modes. Transport mode authenticates the payload of the original IP packet, adding the Authentication Header (AH) between the IP header and the payload. Tunnel mode encapsulates the entire original IP packet inside a new IP packet and therefore authenticates the entire original (now inner) IP packet. In other words, tunnel mode protects the entire IP packet, while transport mode does not.

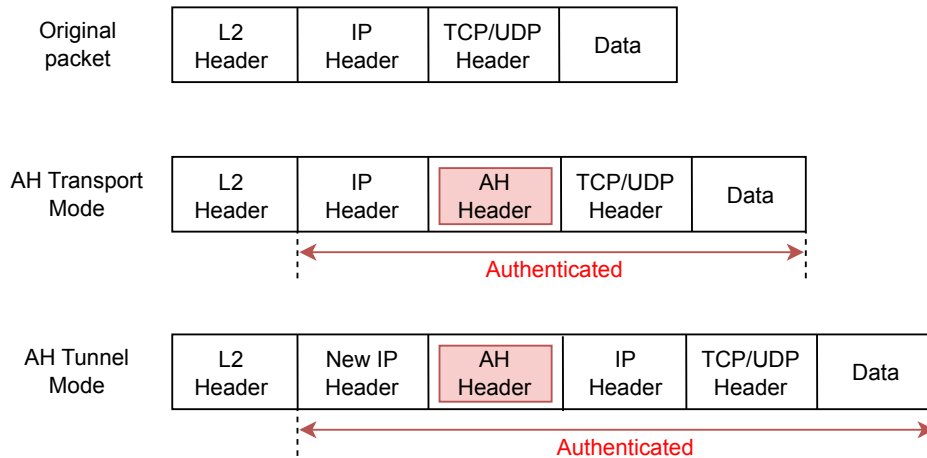


Figure 1.9: Illustration of Transport and Tunnel modes for the Authentication Header (AH).

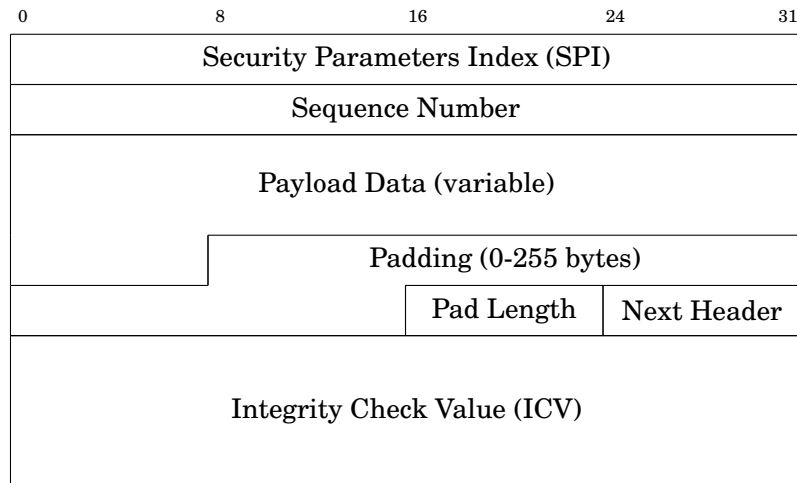


Figure 1.10: Encapsulating Security Payload (ESP). [178]

1.2.6 Encapsulating Security Payload (ESP)

Encapsulating Security Payload (ESP), also part of the IPsec [179] framework, is used to provide what the Authentication Header (AH) already provides, plus confidentiality (*i.e.*, encryption).

Fig. 1.10 shows the header format of Encapsulating Security Payload (ESP), as specified by RFC 4303, with the definition of the following fields:

- **SPI (32-bit)** Arbitrary value that is used by a receiver to identify the Security Association (SA) to which an incoming packet is bound.
- **Seq. Number (32-bit)** Contains a counter value (per SPI) that increases by one for each packet sent.
- **Payload Data** Variable-length field that contains data from the original IP packet.

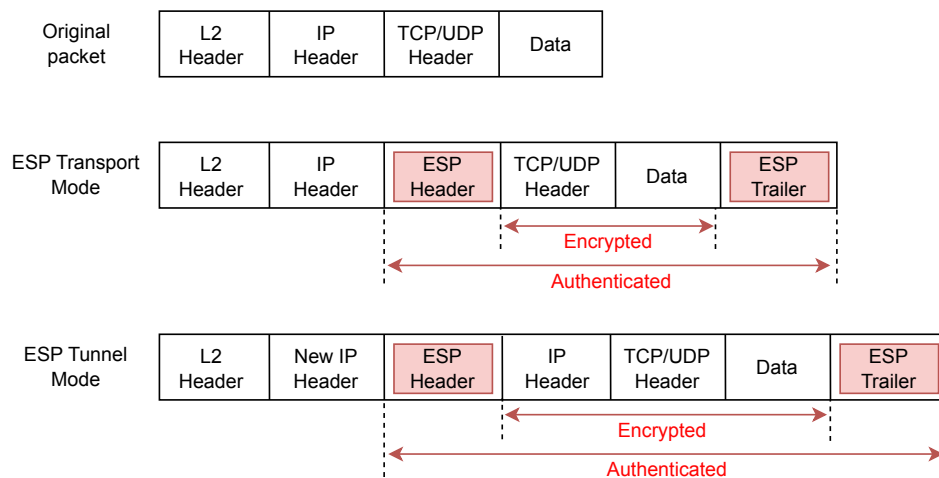


Figure 1.11: Transport and Tunnel modes for Encapsulating Security Payload (ESP).

- **Padding** Variable-length field, 0 to 255 bytes of padding.
- **Pad Length (8-bit)** Number of pad bytes in the Padding field.
- **Next Header (8-bit)** Identifies the type of data contained in the Payload Data field.
- **ICV** Variable-length field that contains the Integrity Check Value for the packet.

The theoretical lower bound for the Encapsulating Security Payload (ESP) size is 10 bytes, with no payload, no padding, and no ICV (which is highly unlikely). The theoretical upper bound for the Encapsulating Security Payload (ESP) size is bigger than 65,535 bytes, although it must fit within the path MTU. In practice, both lower and upper bounds can vary a lot based on the different variable parameters that come into play.

1.2.6.1 Transport and Tunnel modes

Encapsulating Security Payload (ESP) can work in two different modes: transport and tunnel modes. Fig. 1.11 illustrates the difference between these two modes. Transport mode encrypts and authenticates the payload of the original IP packet, adding the ESP header between the IP header and the payload. Tunnel mode encapsulates the entire original IP packet inside a new IP packet and therefore encrypts and authenticates the entire original (now inner) IP packet.

1.2.7 Others

In addition to the Extension Header types described above, the IANA's IPv6 Extension Header Types registry [143] also specifies three others: Mobility Header, Host Identity Protocol Header, and Shim6 Protocol Header. This document only provides a high-level overview for each of them,

for the sake of completeness, as they are considered special Extension Headers and are pretty much dead.

The Mobility Header [237] (protocol number 135) is used in Mobile IPv6, which allows a device to move across networks while maintaining session continuity with its correspondent nodes. It does not expect an upper-layer protocol after itself.

The Host Identity Protocol Header [218] (protocol number 139) allows consenting hosts to securely establish and maintain shared IP-layer state, allowing separation of the identifier and locator roles of IP addresses, thereby enabling continuity of communications across IP address changes. It does not expect an upper-layer protocol after itself.

The Shim6 Protocol Header [227] (protocol number 140) provides multihoming support by allowing IPv6 hosts to maintain connections across multiple addresses, enabling failover and locator agility without requiring changes to the transport layer, through the use of a layer-3 shim sub-layer between IP and transport protocols. It does not expect an upper-layer protocol after itself for Control messages, which is different for the Payload Extension format where an upper-layer is possible.

1.3 Challenges related to Extension Headers

Despite the flexibility and extensibility that IPv6 Extension Headers provide, their use in real-world deployments faces several significant challenges [59, 110, 124, 139, 140, 193, 290], in particular across the global Internet. While Extension Headers can be used reliably within private or controlled environments (*i.e.*, so-called *limited domains*), their presence in IPv6 packets traversing public networks often leads to inconsistent and problematic behavior. Indeed, a growing number of studies show that IPv6 packets carrying one or more Extension Headers are frequently dropped on transit. These studies and related works will be discussed in the next chapter.

The first obvious and most likely cause of such drops is due to policies applied by operators. The reason can be twofold: security [124] and performance [110]. The Hop-by-Hop Options Header is a perfect example of this. For instance, network administrators may configure a router or firewall to block IPv6 packets with Extension Headers, just because they do not need them, following the rule “*What you do not need, you disallow*”. Doing so, they mitigate potential security vulnerabilities and reduce processing overhead in fast-path forwarding. While this behavior is understandable for stub networks, it becomes problematic when dealing with transit networks.

To some extent, such drops may also be due to hardware limitations (*i.e.*, old hardware). Routers usually need to parse the upper-layer (*e.g.*, Layer-4 ports) for filtering or to perform a lookup. However, with the presence of Extension Headers in a packet, the upper-layer header is pushed further, and some routers may not have a parsing buffer large enough to hold the entire packet header. In the best case, which is highly unlikely, the packet is punted to the slow path. In

the worst case, which is most likely, the packet is dropped. Older (legacy) routers are known [49] to have small parsing buffers, and it is difficult to assess the percentage of these routers among the global fleet of routers across the world. This problem applies not only to routers, but also to all network devices such as security devices (*e.g.*, firewalls).

A third cause of such drops may come from complex architectures [76, 78] (*e.g.*, cloud providers). A fourth cause, to some extent, may come from implementation bugs [75], although less likely.

Moreover, Extension Headers also raise operational considerations. For example, operators must ensure that inserting Extension Headers into packets does not cause them to exceed the path MTU. Otherwise, this would open the Pandora's box of fragmentation combined with Extension Headers, which nobody wants for obvious performance and security reasons. Or worse, this could result in total packet loss if fragmentation is not allowed and the system relies on a specific MTU value. Then, regarding load balancing or Quality of Service (QoS) in general, a consequence of Extension Headers is that packets may no longer receive the same treatment as the original packets without Extension Headers (*e.g.*, path change). Therefore, all these considerations must be taken into account and addressed appropriately when using Extension Headers.

RELATED WORK

This section covers work related to IPv6 Extension Headers, from both standardization and research points of view. Regarding standardization, the IETF has published thousands of RFCs. Some of these have significantly influenced the specification, processing behavior, and operational deployment of Extension Headers. The research community has also played a crucial role. Several studies and measurements provide a better understanding of the general state of Extension Headers on the Internet.

2.1 Standardization efforts

Fig. 2.1 represents the timeline of published IETF RFCs that are related to IPv6 Extension Headers. **RFC 1883** [62], published in 1995, served as the foundational specification for IPv6 and its Extension Header mechanism, until it was obsoleted by **RFC 2460** [63] in 1998, which was in turn obsoleted by **RFC 8200** [64], the new IPv6 standard published in 2017. While RFC 2460 required that all nodes *must* examine and process the Hop-by-Hop Options Header, RFC 8200 changed that requirement to a *may*, so that transit nodes only examine and process the Hop-by-Hop Options Header if explicitly configured to do so. RFC 8200 also resolves a lot of security issues introduced by RFC 2460, especially related to the Fragment Header and based on RFC 6946 [107], RFC 7112 [113], and RFC 8021 [112]. **RFC 6564** [191] provides a common format for defining new IPv6 Extension Headers, while **RFC 7045** [40] clarifies how intermediate nodes should deal with Extension Headers. **RFC 8504** [44], the new specification for IPv6 node requirements which obsoletes RFC 6434 [169], clarifies processing rules for packets with Extension Headers, and allows hosts to enforce limits on the number of options included when it applies. **RFC 9098** [109] summarizes the operational implications of Extension Headers and attempts to analyze reasons why packets with Extension Headers are often dropped in the public

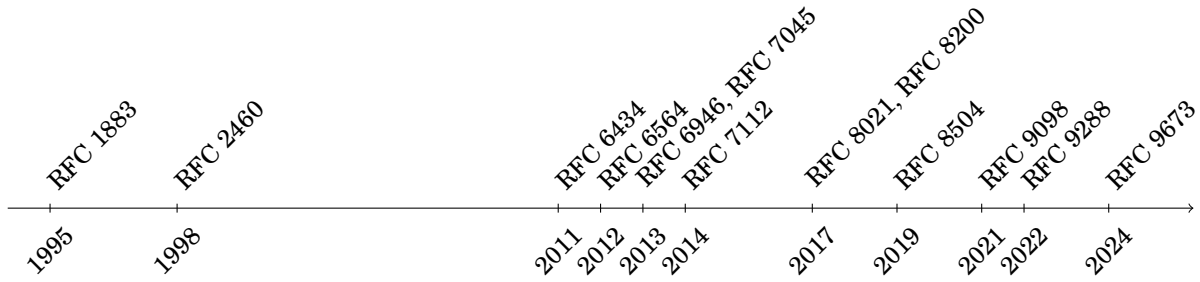


Figure 2.1: Timeline of IETF RFCs related to Extension Headers.

Internet, while **RFC 9288** [111] discusses the operational and interoperability implications of discarding packets based on Extension Headers and provides advice on the filtering of such packets on transit routers. More recently, **RFC 9673** [131] modifies the procedures specified in RFC 8200 to make the Hop-by-Hop Options Header useful to deploy on routers and hosts.

2.2 Studies and Measurements

Gont et al. [110] observe how an 8-byte Hop-by-Hop Options Header, an 8-byte Destination Options Header, and a Fragment Header survive over the Internet. Gont et al. perform traceroute measurements towards servers belonging to the Alexa top 1M domains and find that such Extension Headers are often dropped in transit networks. **Hendriks et al.** [124] state that dropping all traffic containing any Extension Header is the de facto rule applied by operators, for security reasons. To support their claim, they perform limited measurement campaign on a national research network (CSNET) and a campus network (UTNET). In the same spirit, **Padurean et al.** [286] run large-scale traceroute measurements to find the presence of Segment Routing [91, 92] with IPv6 as forwarding plane. They reported no presence of such a deployment, probably due to Extension Headers filtering according to them. **Elkins et al.** [75] propose a methodology for isolating the reasons and network devices responsible for Extension Headers drops. In particular, they discuss a situation in which a tested server is behind a Content Delivery Network (CDN). However, they do not perform any measurements. **Elkins et al.** [74] also focus on the Performance and Diagnostic (PDM) Destination Options Header Option [77]. Such an Extension Header option provides sequence numbers and timing information as a basis for measurement. They do not report any drop when measurements are performed between hosting services while they observe some drops when measurements are sent towards Alexa top 1M domains. **Huston and Damas** [139, 140] report an improvement, over the years, in processing the IPv6 Fragment Header. They also notice that the Destination Options Header and Hop-by-Hop Options Header are generally not supported on public Internet infrastructure. **Custura et al.** [59] present an extensive measurement campaign with a focus on access and server edge networks, and provide results indicating the traversal across Internet paths of packets that include either a Hop-by-Hop

Table 2.1: Comparison of related work with the contributions of this document. An empty-cell means no support, “~” means partial support, and ✓ means full support.

Experiment	Work reference									This document
	[110]	[124]	[286]	[75]	[74]	[139, 140]	[59]	[193, 290]	[234]	
Hop-by-Hop Options Header	~	~				✓	✓	✓	✓	✓
Destination Options Header	~	~			~	✓	✓	✓	✓	✓
Fragment Header	~	~				✓		✓		✓
Routing Header		~	~					✓		✓
Authentication Header (AH)		~						~		✓
Encapsulating Security Payload (ESP)		~						~		✓
Mobility Header										✓
Host Identity Protocol Header		~								✓
Shim6 Protocol Header		~								✓
Large-scale measurement	✓	~	✓		✓	✓	✓	✓		✓
Router isolation									✓	✓

Options Header or a Destination Options Header. Their results indicate that successful reception across an IPv6 path can currently depend on the type of included Extension Headers, their sizes, and on the transport protocol. **Léas et al.** [193, 290] test a large set of Extension Headers in full mesh through multiple vantage points, with some of them located in distinct Autonomous Systems (ASes). By definition, both source and destination of measurements are controlled. Among others, Léas et al. report that path traversal diminishes as the size of Extension Headers increases. Finally, **Ouellette** [234] tests the forwarding of the Hop-by-Hop Options Header and Destination Options Header on six different routers. Ouellette notices that one router fails in forwarding packets in certain scenarios (*i.e.*, packets with Extension Headers of 256 and 512 bytes), which is likely due to hardware limitations.

Table 2.1 summarizes the contributions of related work and compares them with the contributions of this document. An empty-cell means no support, “~” means limited support, and ✓ means full support. As shown in the table, none of the related work measures at large-scale the entire set of Extension Headers with full support (*e.g.*, different sizes). Also, only one performs router isolation to find suspicious behaviors with Extension Headers but, unfortunately, measurements are pretty limited. In this document, we go further by testing all Extension Headers currently defined, *i.e.*, Hop-by-Hop Options Header, Destination Options Header, Fragment Header, Routing Header, Authentication Header (AH), Encapsulating Security Payload (ESP), Mobility Header, Host Identity Protocol Header, and Shim6 Protocol Header, based on a tool we developed called FISHNET. Each of these Extension Headers is tested with different sizes when it applies. All vantage points are unique, *i.e.*, hosted by different Cloud Providers and running in distinct locations, therefore evaluating the edge and core of the Internet. We also provide observations in a controlled environment, comparing the behavior of routers from different vendors when processing Extension Headers, and we investigate their impact on the RTT and path length. Overall, this document covers all aspects and is more comprehensive than related work.

Extended BPF (eBPF [269]), also commonly referred to as BPF (Berkeley Packet Filter) nowadays, is a technology with origins in the Linux kernel (which is now also available [203, 204] for Windows [206]) that can run untrusted programs in a privileged context. Originally, BPF was used for capturing and filtering network packets. eBPF is the evolved form which is more powerful, more flexible, and offers more features such as dynamic tracing and kernel-level programming. Fig. 3.1 illustrates how eBPF works. With eBPF, one can dynamically program the kernel for efficient networking, observability, tracing, and security, thanks to sandboxed programs that run within the operating system. In other words, application developers can run eBPF programs to add capabilities to the operating system at runtime. There is also eXpress Data Path (XDP) [60, 133], which is a subset of eBPF used for networking and packet processing specifically. It has specialized eBPF hook points in NIC drivers, before the kernel allocates buffers. The goal is to quickly handle packets before they enter the kernel network stack. For example, XDP is useful for use cases like DDoS mitigation and packet filtering (*e.g.*, L4Drop [17, 81] from Cloudflare), load balancing (*e.g.*, Katran [149, 255] from Meta), and many others. XDP is also available [205] for Windows.

Recent years have seen a growing interest in using eBPF to deploy network services. For example, Tran and Bonaventure [282] leveraged eBPF to support use cases through injection of options in Multipath TCP (MPTCP). Xhonneux et al. [300] developed a framework allowing network operators to encode their own network functions as eBPF code that is automatically executed while processing specific packets. More recently, pwru [215, 265] (*i.e.*, “Packet, where are you?”) provides a way to trace network packets in the Linux kernel with advanced filtering capabilities. It allows fine-grained introspection of kernel state to facilitate debugging network connectivity issues. Another example is hyDNS [14], a DNS accelerator based on a recursive resolver

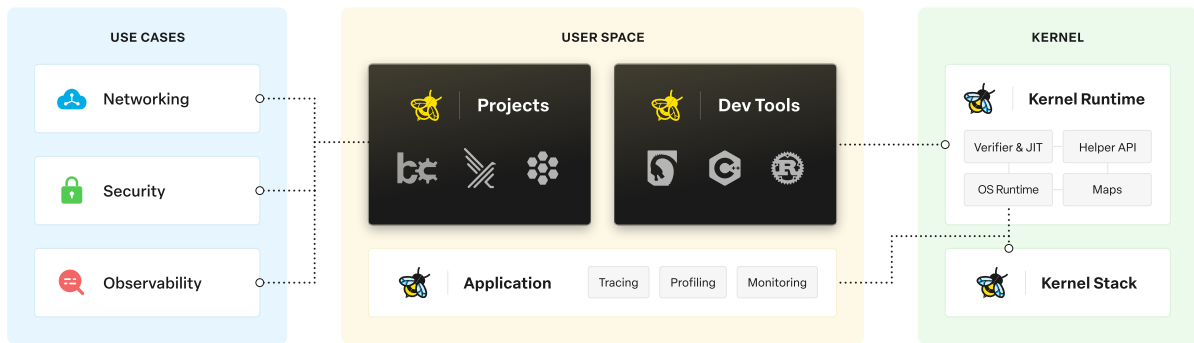


Figure 3.1: Overview of eBPF. [269]

```

1 # tc qdisc add dev eth0 clsact
2 # tc filter add dev eth0 egress proto ipv6 bpf da obj ebpf_program.o sec section

```

Listing 3.1: Commands to attach an eBPF program to an interface on egress.

that combines eBPF offloading in the kernel with a user-space resolver. The DNS kernel cache allows most requests to be served before reaching the kernel network stack. Overall, many other open-source applications are available [269]. eBPF has also found usage in telemetry [132], load balancing in clouds [186], and security [299]. Even more recently, eBPF has found usage in Machine Learning [45, 258].

Relying on the increasing interest in eBPF, this chapter presents a project that leverages eBPF for injecting IPv6 Extension Headers. In particular, the following contributions are provided:

- **eBPF IPv6 Extension Headers Injection (FISHNET)** [156] is introduced, which is a new eBPF program that allows to inject any Extension Header in network traffic. With FISHNET, it is easy to inject dedicated Extension Headers in existing traffic, without modifying end-hosts.
- Performance of FISHNET is evaluated and the results are analyzed for different Extension Header sizes.
- A specific use case is discussed as an example, in which FISHNET would find a suitable use. This example demonstrates the need for operators to ensure they can use a specific Extension Header prior to large-scale deployment of services.
- Source code is available [156] to the research community.

3.1 Implementation

In order to inject one or multiple Extension Headers in outgoing traffic using eBPF, FISHNET must be attached to an interface. More specifically, one needs to add a `clsact qdisc` [197] to

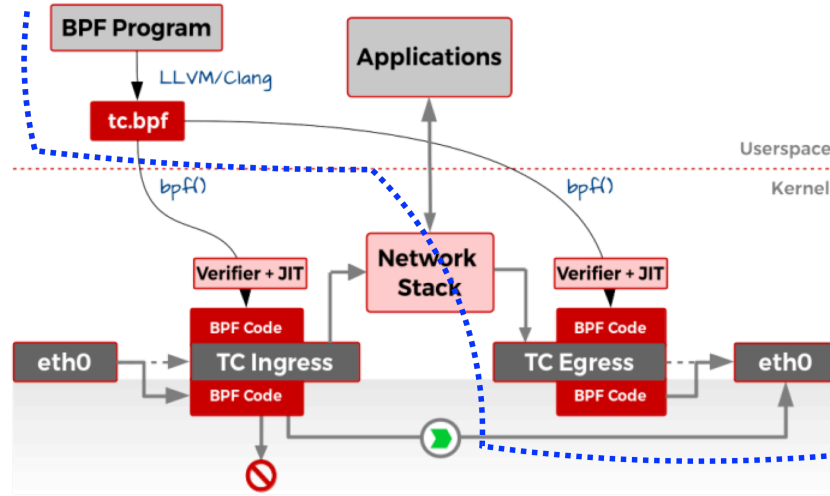


Figure 3.2: Overview of how TC works with eBPF [283].

```
Usage: ./build/tc_ipv6_eh_user.o { --disable | --enable [ --force ]
EXTHDR [ EXTHDR ... EXTHDR ] }

EXTHDR := { --hbh 8..2048 | --dest 8..2048 | --rh0 24..2040 | --rh2
| --rh3 24..2040 | --rh4 24..2040 | --fragA | --fragNA | --ah 16..10
24 | --esp 16..2048 }

If a size is required, it MUST be an 8-octet multiple.
Routing Header sizes minus 8 MUST be 16-octet multiples.

Accepted chaining order, as per RFC8200 sec4.1:
- Hop-by-Hop Options header
- Destination Options header
- Routing header
- Fragment header
- Authentication header
- Encapsulating Security Payload header
- Destination Options header
```

Figure 3.3: FISHNET commands (user program).

an interface (Listing 3.1, line 1), which is a special queuing discipline that does not perform traffic shaping but instead provides hook points for classifiers and actions in both ingress and egress directions. Then, FISHNET must be attached to an egress filter on that interface (*i.e.*, the TC [268] egress hook), where the eBPF program will be executed, with a specific section to be run (Listing 3.1, line 2). Both commands use TC, a traffic control tool as part of the iproute2 [267] suite. Finally, the user program is used to configure the Extension Headers injection. Fig. 3.2 provides a high-level picture of how it works. What was previously described is represented on the right side of the dashed blue line.

Listing 3.2 shows the simplified eBPF kernel program of FISHNET to inject a buffer of bytes in egress traffic. The program is totally agnostic of the content of the buffer, which is built by the user program (see below). Lines 1 to 7 represent the data structure used to inject Extension

```

1 struct exthdr_t {
2     __u8 nh;
3     __u32 last_nh;
4     __u32 len;
5 #define MAX_BYTES 2048 /* Increase as you wish */
6     __u8 bytes[MAX_BYTES];
7 };
8
9 struct {
10     __uint(type, BPF_MAP_TYPE_ARRAY);
11     __uint(max_entries, 1);
12     __type(key, __u32);
13     __type(value, struct exthdr_t);
14     __uint(pinning, LIBBPF_PIN_BY_NAME);
15 } map SEC(".maps");
16
17 SEC("egress")
18 int eh6(struct __sk_buff *skb)
19 {
20     __u32 off, idx = 0, mtu = 0;
21     struct exthdr_t *eh;
22     struct ipv6hdr *ip6;
23     __u8 nh;
24
25     ip6 = ipv6_header(skb, &off);
26     if (!ip6)
27         return TC_ACT_OK;
28
29     eh = bpf_map_lookup_elem(&map, &idx);
30     if (!eh)
31         return TC_ACT_OK;
32
33     if (eh->len < 8 || eh->len > MAX_BYTES || bpf_check_mtu(skb,
34         skb->ifindex, &mtu, eh->len, 0))
35         return TC_ACT_OK;
36
37     nh = ip6->nexthdr;
38     ip6->nexthdr = eh->nh;
39     ip6->payload_len = bpf_htons(bpf_ntohs(ip6->payload_len) + eh->len);
40
41     if (bpf_skb_adjust_room(skb, eh->len, BPF_ADJ_ROOM_NET, 0))
42         return TC_ACT_SHOT;
43
44     if (bpf_skb_store_bytes(skb, off, eh->bytes, eh->len, 0))
45         return TC_ACT_SHOT;
46
47     if (eh->last_nh < MAX_BYTES && bpf_skb_store_bytes(skb, off +
48         eh->last_nh, &nh, sizeof(nh), 0))
49         return TC_ACT_SHOT;
50     return TC_ACT_OK;
51 }

```

Listing 3.2: Simplified eBPF kernel program to inject a buffer of bytes in egress traffic.

Headers, which is stored in a map (defined by lines 9 to 15). The map is pinned by name so that the user program is able to interact with it. This is also why the map is not defined per-CPU, since the user program needs to update the map (which is not possible for each CPU from user space). The function in line 18 is the handler for the *egress* section, and performs the following steps: (i) checks for a pointer overflow¹, which is required by the verifier (line 25); (ii) fetches the map (line 29); (iii) applies some boundary checks required by the verifier and makes sure the insertion will not exceed the MTU (line 33); (iv) updates the IPv6 Next Header field and payload length according to what is inserted (lines 36 to 38); (v) makes room for the new bytes (line 40); (vi) injects the new bytes in the packet (line 43); (vii) updates the Next Header field of the last Extension Header to the upper-layer protocol used in the current packet (line 46), so that the Extension Header chain is complete.

Overall, the FISHNET piece of code described in Listing 3.2 is completely agnostic of whether one or more Extension Headers are injected, or their order. The only thing that it knows is that it has to inject a buffer of bytes. Therefore, the overhead only depends on the number of bytes to inject. Indeed, the buffer construction is delegated to the user program which is responsible for configuring what will be injected (one or more Extension Headers, their order, etc). Fig. 3.3 shows the “--help” output of the user program running on top of FISHNET to illustrate how it works and its possibilities. Very briefly, Extension Headers can be injected with constraints on respective sizes, and in any order. If the chosen order does not respect RFC 8200, an error is returned as a security barrier. Should the user really want to do it no matter what, the “--force” flag can be passed to override the restriction. For example, as soon as the eBPF kernel program (see Listing 3.2) is attached to a specific interface (see Listing 3.1), passing “--enable --hbh 64” to the user program (see Fig. 3.3) would trigger the eBPF kernel program for each outgoing packet on that interface and inject a 64-byte Hop-by-Hop Options Header into each packet.

3.2 Performance evaluation

This section describes the methodology to evaluate FISHNET (Sec. 3.2.1) and discusses the performance results (Sec. 3.2.2).

3.2.1 Methodology

To evaluate FISHNET, we rely on TREX [48], an open source, low cost, stateful and stateless traffic generator fueled by DPDK [273]. It has multiple advantages, such as the ability to generate Layer3–7 traffic and multiple streams, as well as the ability to easily craft specific packets with its underlying Scapy [252] layer.

Fig. 3.4 illustrates the straightforward testbed: one machine for TREX, and another one for the *Device Under Test* (DUT). Both are equipped with an *Intel XL710 2x40GB QSFP+* Network

¹Dynamic pointers (*dynptrs*) were considered but finally not used due to how the program is designed.

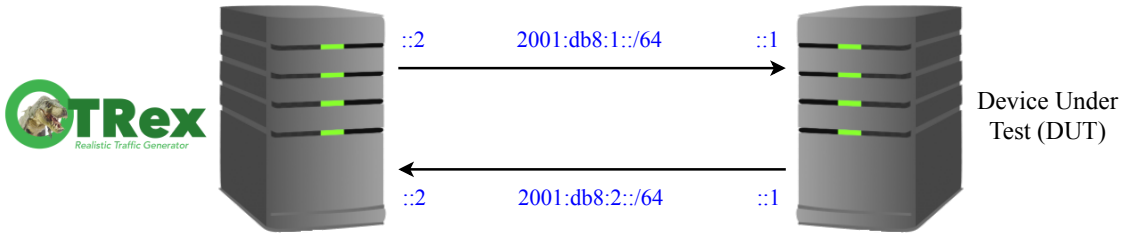


Figure 3.4: Testbed with TReX to evaluate FISHNET.

Interface Card (NIC), each connected port to port 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 isolate a specific function on the DUT and evaluate it, *i.e.*, the egress injection of IPv6 Extension Headers with FISHNET in this case. The DUT has an *Intel Xeon e5-2630 v3 at 2.40GHz* Central Processing Unit (CPU), with 8 Cores, 16 Threads, and has 16GB RAM. It runs a Linux kernel version 6.9.0 and FISHNET was compiled with clang [276] version 14.0.6. Iproute2 was compiled with libbpf [275] version 1.4.0. 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. The MTU is set to 2,148 so that the maximum size injected (*i.e.*, 2,048 bytes) would not make packet sizes to exceed it. 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).

As explained in Sec. 3.1, FISHNET is completely agnostic of the content of the buffer, and only the number of bytes to inject may have an impact on performance, whatever the combination of Extension Headers. Since most of these Extension Headers are limited to a maximum of 2,048 bytes (except for ESP), we evaluate the impact of an injection from 0 to 2048 bytes, although a combination of Extension Headers could result in a much bigger buffer. In that case, one could simply increase MAX_BYTES even more in FISHNET. Note that the kernel prevents `bpf_skb_adjust_room` from adding more than 4,096 bytes.

3.2.2 Results

Fig. 3.5 shows the impact on throughput of different injection sizes. The forwarding baseline of the DUT is approximately 1.195.000 packets per second (pps) on a single core. One can directly observe a loss of 19% (*i.e.*, approximately 225.000 pps) when injecting the minimum size of 8 bytes, which then remains stable up to 128 bytes. The fact that such a loss occurs immediately will be discussed below. When injecting 256 bytes or more, the loss rate bumps to 58%. This huge drop is due to a lack of space in the packet headroom, which involves implicit reallocation by the kernel to make the headroom larger. Headroom space varies depending on the CPU architecture and, in

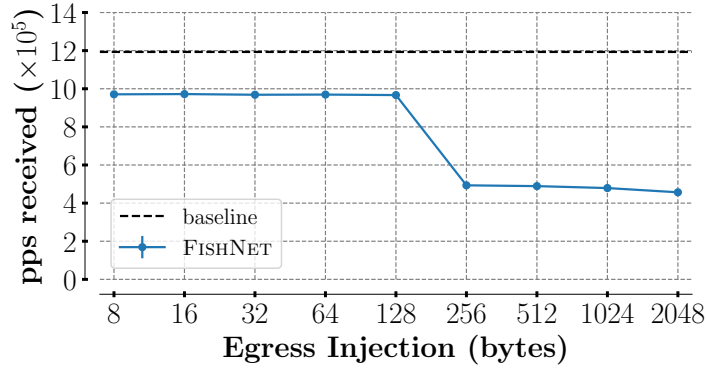


Figure 3.5: IPv6 Extension Headers injection with FISHNET and its impact on throughput.

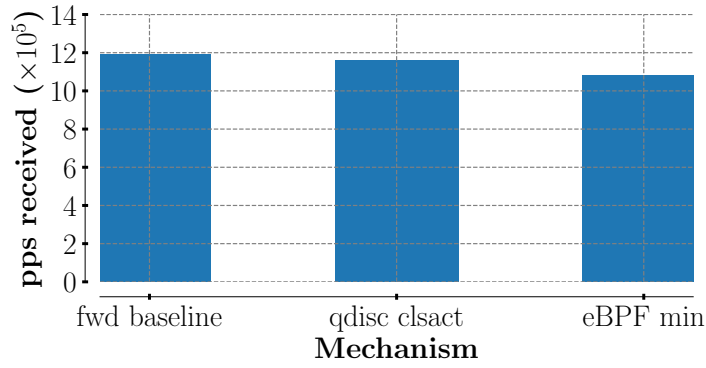


Figure 3.6: Impact on throughput between adding a `clsact qdisc` [197] and running a minimal version of a TC/eBPF program on egress.

particular, the NIC driver. In our case, *i.e.*, `x86_64` architecture and `i40e` driver, the headroom has an initial size of 192 bytes² [274]. This means that as soon as we inject 193 bytes or more, the performance drop will happen. This “bug” will be discussed in more detail and solved in the second part of this document.

Fig. 3.6 shows the impact on throughput when only a `clsact qdisc` is added (see Listing 3.1, line 1), and it is compared to when a minimal³ eBPF kernel program is running. One can see that only adding the `qdisc` gives a 3% loss already, while running a minimal eBPF kernel program gives an additional 7% loss, which makes it a total of 10% loss. It would be interesting to investigate if improvements can be made on that part. In fine, the real loss rate of our eBPF kernel program is 9%, based on this initial and unavoidable 10% loss.

Another interesting observation is that some network drivers (*e.g.*, `e1000e`, `vmxnet3`) seem to have issues with checksum offloading⁴ when there is at least an Extension Header in packets,

²With `legacy-rx` disabled (by default)

³Minimal means that the section handler directly returns `TC_ACT_OK`.

⁴Checksum offloading means that the NIC performs the calculation of checksums instead of the host CPU.

which is a problem totally independent of FISHNET. Indeed, the checksum calculated in Layer-4 (*i.e.*, the upper-layer) is sometimes incorrect, even if adding Extension Headers should have no impact since it does not modify the pseudo-header⁵. The only exception is with a Routing Header, where the destination in the pseudo-header is the last segment. Because of this bug, packets may disappear along the path, which could falsely suggest a Layer-2 problem that is hard to debug. This should be investigated as well, and some work [127] has already begun to fix the bug related to checksums and IPv6 Extension Headers in the Linux kernel.

Finally, it is worth mentioning that we evaluate the worst case, *i.e.*, line rate traffic on a single core. Overall, it is highly unlikely that an operator would need FISHNET to inject Extension Headers at line rate in order to simulate and test a service (even less with a single core). However, independently of FISHNET and for use cases in production, some operators may still want to inject Extension Headers in line-rate traffic. The initial cost shown in Fig. 3.6 could prevent them from following that path.

3.3 Use case example

Let us consider an operator wanting to implement and deploy a new service that would require the use of large DNS requests and replies (*i.e.*, sometimes above 1,280 bytes, sometimes up to 1,500 bytes and more). In IPv6, the minimum link MTU is 1,280 bytes (although 1,500 is quite common), which means that bigger packet sizes are exposed to fragmentation, depending on the path MTU. The IPv6 fragmentation is always performed by the source and uses the Fragment Header. Based on some recent studies [9, 110, 138–140, 168, 193, 290], the operator figures out that the Fragment Header does not always survive over the global Internet, which is problematic considering the service. However, those studies do not necessarily focus on the Fragment Header for the DNS service. Therefore, before implementing the entire new service and in order to avoid wasting time foolishly, the operator wants to make sure that all fragmented DNS requests and replies would go through and reach the target. On top of that, the operator would also like to use the Minimum Path MTU Hop-by-Hop Option [130] in order to detect the minimum path MTU. The aforementioned studies show that even a small Hop-by-Hop Options Header almost never survives over the global Internet. FISHNET could be a fast way of testing the service, *i.e.*, an easy way to develop a proof-of-concept, by injecting either a Fragment Header or a Hop-by-Hop Options Header to some specific packets. There are two possible outcomes: either (*i*) the test is successful, and so the operator can confidently implement and deploy its service; or (*ii*) the test is not successful, and so the operator needs to think about an alternative for its new service.

⁵The pseudo-header is used to calculate checksums for upper-layer protocols like TCP or UDP, and is specified in RFC 8200 Sec. 8.1.

3.4 Possible improvements

FISHNET is already useful as it is for operators to quickly develop proof-of-concepts based on Extension Headers and for researchers to perform measurements on Extension Headers. However, it could still benefit from several improvements:

- Add a feature to be able to pass a configuration file to the user program in order to have some specific and pre-configured Extension Headers data (*e.g.*, based on a YANG [20] model). It would be needed for use cases that require real data.
- Add another feature to have per-packet “post-processing”, depending on the Extension Header type (*e.g.*, to have real encryption with ESP). This feature would probably reduce performance but such a trade-off may be required in some cases.
- Investigate the recent `tcx` addition [23] as an alternative to `clsact` `qdisc` to reduce the initial overhead observed in Fig. 3.6.

EXTENSION HEADERS SURVIVABILITY IN A CONTROLLED ENVIRONMENT

In order to be able to answer the first Research Question (**RQ-1**), namely “*Can we actually use IPv6 Extension Headers on the Internet?*”, we first need to understand how routers handle IPv6 Extension Headers. This chapter investigates the capacity of IPv6 packets with Extension Headers to traverse a single router and arrive unmodified at the destination, in a controlled environment. The goal of this experiment is to verify whether recent (and not so recent) routers have any hardware limitations related to the parsing of Extension Headers. Sec. 4.1 describes the infrastructure for the experiment, while Sec. 4.2 discusses the results.

4.1 Infrastructure Setup

Table 4.1: List of tested routers in our controlled environment.

	Vendor	Model	Version
R1	Cisco	ASR1001-X (ASIC ¹ based)	IOS XE 03.16.05.S
R2			IOS XE 17.06.04
R3		ASR9904 (NPU ² based)	IOS XR 7.9.21
R4	Huawei	AR617VW-LTE4EA	V300R019C10
R5	Juniper	vMX	Junos OS 20.2R1.10
R6	Linux	–	Kernel 6.11
R7	Nokia	7750 SR-7	20.10

¹ASIC: Application-Specific Integrated Circuit

²NPU: Network Processing Unit

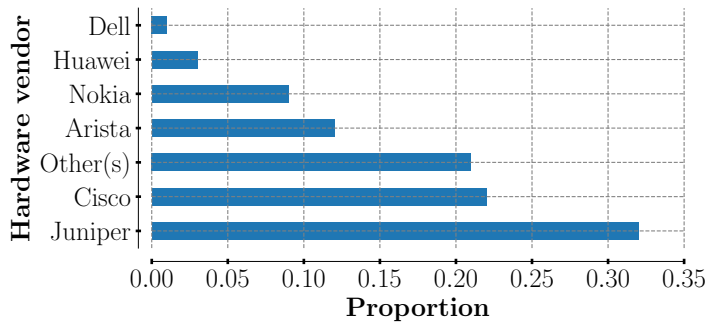


Figure 4.1: Distribution of hardware vendors in the survey results.

Our controlled environment is made of two devices (*i.e.*, the sender and the receiver) and a physical router (*i.e.*, the DUT, a real hardware) in between them. We evaluate seven different routers (DUTs) separately, each of them running a default IPv6 configuration. Those routers are listed in Table 4.1 and were deliberately chosen based on the responses received from our survey to operators.³ In the table, only the Cisco and Nokia routers use dedicated hardware to process packets (*i.e.*, ASICs or NPUs), which imposes some hardware-specific limitations (*e.g.*, fixed-size parsing buffers). Such limitations generally do not apply to other routers in the table that rely solely on a general-purpose host CPU. Similarly, the notions of fast path and slow path are usually related to routers using dedicated hardware: fast path means that the packet is processed by the hardware, while slow path means that the packet is processed by the software (*i.e.*, the general-purpose host CPU). However, a router without dedicated hardware could also share the same notions of fast path and slow path, although slightly different (*i.e.*, both in software, with fast path being accelerated or optimized processing).

Fig. 4.1 shows the various hardware deployed by operators, according to the survey. Both Juniper and Cisco routers seem to share the biggest part of the market [2, 3, 201], followed by Linux or FreeBSD (with BIRD or FRRouting running on top – the “Other(s)” category in Fig. 4.1), followed by Arista, Nokia, Huawei, and Dell. Note that Mikrotik and Ubiquity also appeared in the “Other(s)” category in Fig. 4.1. Obviously, we were unable to get routers from all possible vendors, but we still managed to get routers from highly represented ones. As for Juniper, and after discussions with their team, it appeared that testing a virtual image of the MX series (for the sake of simplicity, *i.e.*, with Containerlab [56]) was enough compared to a real hardware, since hardware limitations are faithfully simulated according to them. Overall, routers from different vendors are expected to share the same kind of behavior with a **default** IPv6 configuration (*i.e.*, they are supposed to be RFC-compliant and pass some validation tests), which therefore gives a trend or a big picture of what should be observed on the Internet. As a consequence, we assume that our infrastructure is as representative as possible.

³The survey was sent on both RIPE and NANOG mailing-lists on September 4th 2024. At the time of writing, we received 50 responses.

The sender relies on FISHNET (see Chap. 3) to inject Extension Headers in egress traffic. The sender generates TCP and UDP traffic with Netcat [222], and ICMPv6 traffic with ping6 [195]. Traffic is collected on the sender and on the receiver with tcpdump [278]. The MTU is increased on both links to support large Extension Header sizes (*e.g.*, 2,048 bytes or more).

4.2 Results

This section discusses the results based on experiments performed in our controlled environment. What is observed may not reflect the reality in the Internet, although representative enough to provide a trend. That is why we also perform large-scale measurements on the Internet in the next chapter. Table 4.2 (left column) shows all the experiments (*i.e.*, Extension Header types and sizes) performed on the different routers. All existing Extension Headers are tested, except the Routing Header type 1 (Nimrod) as it was never standardized and is now deprecated [1]. Routing Header type 55 is included for testing a Routing Header with an undefined type and see how the routers behave. Each Extension Header with a specific size is tested three times: with TCP, UDP, and ICMPv6.

Table 4.2 (right column) shows the successful experiments (with a ✓) for each router. With a default IPv6 configuration running on the routers, every single Extension Header is successfully forwarded, whatever the size or Layer-4. Some combinations of two or three Extension Headers were also tested and were successful for all the routers (*e.g.*, a Hop-by-Hop Options Header followed by a Destination Options Header, up to $2 \times 2,048$ bytes in total), despite not being in Table 4.2 for readability reasons. Such results suggest that routers now have bigger hardware capabilities when it applies (*e.g.*, much larger parsing buffers, or packet recirculation [109]), which is good news for Extension Headers in general. Indeed, older (legacy) routers are known [49] to have small parsing buffers, even sometimes as small as 64 bytes. For example, Ouellette [234] reports a router that drops IPv6 packets as soon as the total size of Extension Headers is more than 176 bytes, with a default IPv6 configuration running on the router. For privacy reasons, we do not know the vendor or version of the router, but we were told by the author that it was a very old router. Also, routers usually need to parse past the IPv6 header (*e.g.*, Layer-4 ports) for filtering or to perform a lookup. With the presence of Extension Headers after the IPv6 header, the upper-layer header is pushed further and old routers may not have a parsing buffer large enough. As a result, such routers would probably drop the packet. As a comparison, we run the same experiments in Table 4.2 again but, this time, a filter on TCP (destination port 22) is configured on routers. This is to force routers to parse all the headers and therefore make sure a filter or lookup is triggered, testing if there are any hardware limitations (when it applies). With that, the results are exactly the same as with a default IPv6 configuration on routers. This shows that routers are not responsible for dropping packets with Extension Headers by default, except for old routers that may have hardware limitations (*i.e.*, small parsing buffers). It is important to

Table 4.2: Extension Headers survivability in the controlled lab, for each tested router. Each Extension Header is tested with different sizes when it makes sense. All tests are performed three times: one with UDP, one with TCP, and one with ICMPv6. Each ✓ corresponds to a successful test, *i.e.*, when it successfully goes through the router and reaches the destination.

Extension Headers	Routers						
	R1	R2	R3	R4	R5	R6	R7
Hop-by-Hop Options Header (8, 16, 32, 64, 128, 256, 512, 1024, 2048)	✓	✓	✓	✓	✓	✓	✓
Destination Options Header (8, 16, 32, 64, 128, 256, 512, 1024, 2048)	✓	✓	✓	✓	✓	✓	✓
Fragment Header <i>atomic</i> (Fixed size: 8)	✓	✓	✓	✓	✓	✓	✓
Fragment Header <i>non-atomic</i> (Fixed size: 8)	✓	✓	✓	✓	✓	✓	✓
Routing Header <i>Type 0</i> (24, 72, 136, 264, 520, 1032, 2040)	✓	✓	✓	✓	✓	✓	✓
Routing Header <i>Type 2</i> (Fixed size: 24)	✓	✓	✓	✓	✓	✓	✓
Routing Header <i>Type 3</i> (24, 72, 136, 264, 520, 1032, 2040)	✓	✓	✓	✓	✓	✓	✓
Routing Header <i>Type 4</i> (24, 72, 136, 264, 520, 1032, 2040)	✓	✓	✓	✓	✓	✓	✓
Routing Header <i>Unknown Type 55</i> (24, 72, 136, 264, 520, 1032, 2040)	✓	✓	✓	✓	✓	✓	✓
Authentication Header (AH) (16, 32, 64, 128, 256, 512, 1024)	✓	✓	✓	✓	✓	✓	✓
Encapsulating Security Payload (ESP) (16, 32, 64, 128, 256, 512, 1024, 2048)	✓	✓	✓	✓	✓	✓	✓
Mobility Header <i>Type 0, no option</i> (Fixed size: 8)	✓	✓	✓	✓	✓	✓	✓
Host Identity Protocol Header <i>Type 1</i> (Fixed size: 48)	✓	✓	✓	✓	✓	✓	✓
Shim6 Protocol Header (Fixed size: 8)	✓	✓	✓	✓	✓	✓	✓

note, however, that recent routers using dedicated hardware may not have parsing buffers *that* large (*e.g.*, 2048 bytes or more). In this case, packet recirculation likely comes into play. Overall, should we observe drops on the Internet (see next chapter), the main reason would likely be policies applied by operators, although hardware limitations could also be the reason to some extent. Note that the checksum offloading issue described at the end of Sec. 3.2.2 may also affect Extension Headers survivability, since this feature is often enabled by default on end-hosts (*i.e.*,

senders).

Another interesting observation that is worth mentioning is about a specific test where a Hop-by-Hop Options Header is not in first position (*e.g.*, a Destination Options Header followed by a Hop-by-Hop Options Header). Some routers would drop the packet, while some would not. It is due to the fact that RFC 8200 does not use normative language to enforce those requirements. As a consequence, dropping the packet or not in such a situation are both valid options. However, the best approach here is probably to not drop the packet and follow Postel's Law, also known as the Robustness Principle: "*Be liberal in what you accept, and conservative in what you send*". This, for interoperability reasons and to avoid any protocol ossification in general. After all, operators tend to dislike when a router drops packets that do not break normative rules.

Let us conclude this chapter with a final observation, which is that routers do apply different treatment to packets depending on which Extension Headers are present. What we observed is that routers would punt packets to the slow path (when it applies) every time a Hop-by-Hop Options Header is present, while packets with any other Extension Headers would stay on the fast path (when it applies). It all makes sense, since the Hop-by-Hop Options Header is the only Extension Header to be processed by intermediate nodes, while other Extension Headers are only processed by the destination. Nevertheless, this observation can be useful, especially for the Hop-by-Hop Options Header. Indeed, when a packet with Extension Headers goes through the slow path, a router under heavy load (quite frequent for some routers in the core Internet) may drop it.

EXTENSION HEADERS SURVIVABILITY ON THE INTERNET

In Chap. 4, we investigated how IPv6 packets with Extension Headers can successfully go through routers from different vendors. We did not observe any hardware limitations, and all routers passed the tests. As a next step to answer the first Research Question (**RQ-1**), namely “*Can we actually use IPv6 Extension Headers on the Internet?*”, we want to check whether these observations remain true on the public Internet. In other words, we want to verify that IPv6 packets with Extension Headers can survive a complete journey (*i.e.*, from source to destination) over the Internet. If this is not the case, then the goal is to understand why and, if possible, determine the cause (*i.e.*, policies applied by operators, hardware limitations, or any another reason). This chapter provides a comprehensive view on how Extension Headers are processed in the network by transit nodes.

5.1 Measurement Methodology and Data Collection

Multiple *experiments* are conducted, each one being a five-step process run by a Vantage Point (VP) towards other VPs in our measurement infrastructure:

- Step₁ 20 pings towards the destination VP;
- Step₂ 1 vanilla Paris traceroute [12] towards the destination VP;
- Step₃ 5 Paris traceroutes with a given Extension Header towards the destination VP. Extension Headers are injected using FISHNET (see Chap. 3);
- Step₄ 1 vanilla Paris traceroute towards the destination VP;
- Step₅ 20 pings towards the destination VP.

Table 5.1: Extension Header types and sizes tested during a measurement campaign. Each $\times$ corresponds to an experiment, for a total of 38 experiments.

Extension Header		Extension Header Size (Bytes)														
Name	Type	$\emptyset$	8	16	24	32	40	48	56	64	128	256	512	680	1,024	1368
Destination Options Header		$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$			
Hop-by-Hop Options Header		$\times$										$\times$	$\times$			
Fragment Header	atomic	$\times$														
	non-atomic	$\times$														
Routing Header	2	$\times$														
	0				$\times$									$\times$		$\times$
	3				$\times$									$\times$		$\times$
	4				$\times$									$\times$		$\times$
	55				$\times$									$\times$		$\times$
Authentication Header (AH)			$\times$										$\times$		$\times$	
Encapsulating Security Payload (ESP)			$\times$										$\times$		$\times$	
Mobility Header	0	$\times$														
Host Identity Protocol Header	1	$\times$														
Shim6 Protocol Header		$\times$														

Step₃ is the core of an experiment and aims at testing the survivability of a given Extension Header through five consecutive Paris traceroutes between two VPs. To limit the risk of a path from being changed between consecutive traces (*e.g.*, due to load balancing), all Paris traceroute identifiers are kept identical between traces of all experiments. Pings and classic Paris traceroutes before and after Step₃ are there for reachability reasons, *i.e.*, to check whether the experiment destination is reachable, and for comparison, *i.e.*, to detect whether the path or the RTT changes compared to traffic with Extension Headers. Any experiment is thus made of seven Paris traceroutes and 40 pings. We run each experiment around a particular upper-layer protocol, *i.e.*, TCP, UDP, and ICMPv6, leading to 21 Paris traceroutes per Extension Header. The objective is also to see whether the upper-layer protocol has an impact on Extension Headers processing. Finally, data is collected at each VP through `tcpdump`.

For data collection, we conducted several *measurement campaigns*. We define a measurement campaign as a set of experiments, run in full mesh between VPs, considering all possible Extension Headers and possibly varying the size and type when it makes sense. Table 5.1 lists all considered Extension Headers, with their varying parameters. Note that the size $\emptyset$ means that the corresponding Extension Header has a fixed and predefined size. Combining all parameters (*i.e.*, those with $\times$) leads to 38 different experiments. Which means that, for a pair of VPs in one direction, a measurement campaign corresponds to 798 Paris traceroutes (21 traces multiplied by 38 experiments). Table 5.1 also has all Extension Headers tested in Table 4.2, with some strategic choices on sizes. One reason is simply to reduce the measurements execution time (*e.g.*, with only the min, mid and max values for a Routing Header instead of all sizes, fewer sizes tested for a Hop-by-Hop Options Header since it has been reported a low survivability even for small ones – see Chap. 2 –, etc), while another obvious reason is to not exceed the MTU (*e.g.*, sizes of 1,500 bytes and more).

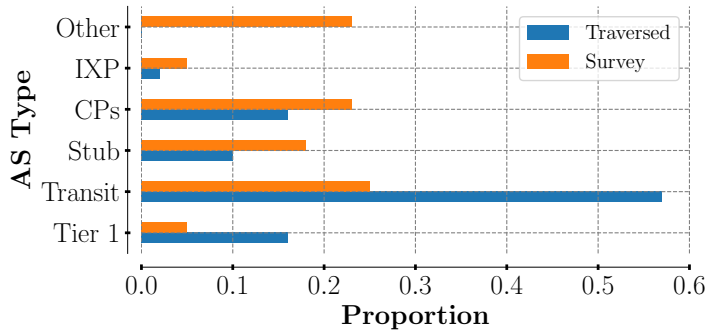


Figure 5.2: Category of ASes traversed by our measurements (total is 64) and survey respondents. Important note: there is no intersection between the list of traversed ASes and the list of ASes that responded to the survey.

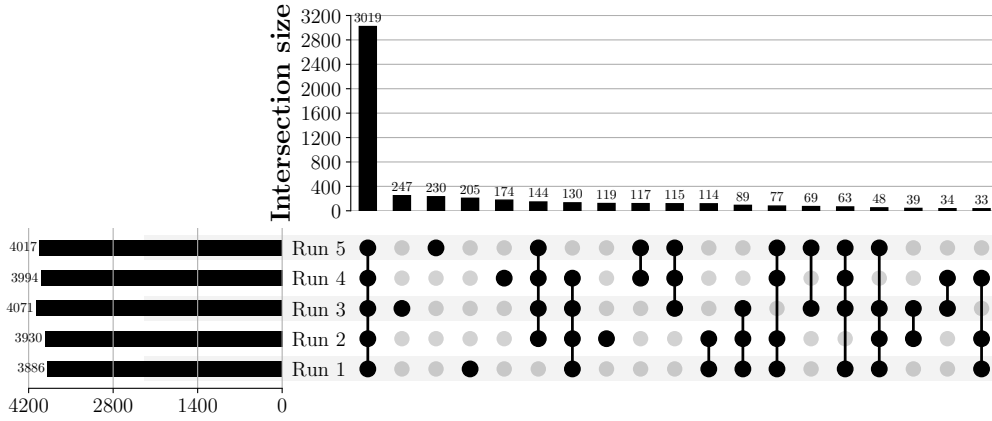


Figure 5.3: UpSet plot – IPv6 addresses collected during Step₂, Step₃, and Step₄ of each Run, and the intersection between Runs.

To run our measurement campaigns, we build a full mesh infrastructure around Cloud Providers (CPs). The reason for this approach is twofold: (i) we wanted full control on destinations in order to capture the received traffic and make sure they are configured to process all Extension Headers correctly, which is not possible with Internet data measurement systems such as RIPE Atlas [247]; and (ii) we were interested in the core/edge point of view, without end-users. Table 5.2 shows the list of considered CPs and the exact location of the virtual machines (VMs) running our experiments, also illustrated on the map in Fig. 5.1. We made efforts in spreading our experiments over distinct CPs to avoid the particular case of inter data-center traffic, with VMs scattered around the world. The majority of the VMs are located in Europe (39.1%), followed by Asia (21.7%). 13% of VMs are located in North America, the same proportion in South/Latin America. Finally, we were able to deploy a few VMs in Africa (8.7%) and Oceania (4.3%).

We run five measurement campaigns (called *Runs* in the following) over two weeks. Each Run roughly lasts 40 hours. This allows us to finally collect 2,018,940 traces ($5 \times 403,788$ traces

per measurement campaign). Fig. 5.3 depicts an UpSet plot [57] illustrating the unique IPv6 addresses we collected during Step₂, Step₃, and Step₄ of each Run and how they intersect. It is another representation for a Venn diagram with a large number of sets (*i.e.*, more than three sets). The figure is made up of three parts: the matrix (bottom right) shows the number of different IPv6 addresses collected in Runs. A dot in the matrix means that at least one IPv6 address has been collected during that Run (*e.g.*, the dot, second column, third row, indicates that some IPv6 addresses were collected during Run 3). If there are multiple black dots on a column, it corresponds to IPv6 addresses collected during multiple Runs (*e.g.*, the first column refers to IPv6 addresses seen in every Run). The histogram on the bottom left is the size of each matrix row, while the histogram on top right gives the number of IPv6 addresses in the corresponding column of the matrix (if we focus on the dot, second column, third row, 4,071 IPv6 addresses were collected during Run 3 – left histogram – with, in particular, 247 of them uniquely observed during Run 3 – top histogram). It is worth mentioning that, for readability reasons, intersection sizes lower than 30 were not included. Run 3 is the one in which we collected most IPv6 addresses (4,071) while the minimum was in Run 1 (3,886). Most of IPv6 addresses (3,019) were observed during every Run. Those addresses are mapped to 64 Autonomous Systems (ASes), which is consistent over the five Runs (relying on ipwhois [239] for lookups against RIR’s databases), most of them (56.25%) being Transit ASes, and 15.62% of them being Tier1 ASes (see Fig. 5.2).

For our measurement campaigns, we implemented RFC 9511 [289] for attribution of Internet probes. In particular, we implemented both in-band and out-of-band recommendations, except for TCP probes where only out-of-band was used to avoid potential packet drops due to the presence of data with a TCP SYN. For in-band, an email address was added to the data payload. For out-of-band, a web server was running on VPs, with the main page being an alias of */.well-known/probing.txt*. That text file described the on-going measurement campaign. So far, no one has contacted us.

5.2 Results

This section describes our results for Extension Headers survivability on the Internet. As discussed in Sec. 5.1, survivability is assessed through Paris traceroute measurements (Step₃).

In Sec. 5.2.1, we provide a general overview of Extension Headers survivability, while Sec. 5.2.2 shows results on a per Extension Header basis. Then, Sec. 5.2.3 discusses where the drop of IPv6 packets with Extension Headers occurs in the network. Finally, Sec. 5.2.4 and Sec. 5.2.5 respectively discuss the impact of Extension Headers on path lengths (*i.e.*, number of hops), and on the round-trip time (RTT).

5.2.1 General Overview

Fig. 5.4 provides a high level overview of Extension Headers survivability between CPs. In particular, it shows the proportion of survivability across the full mesh of CPs, considering

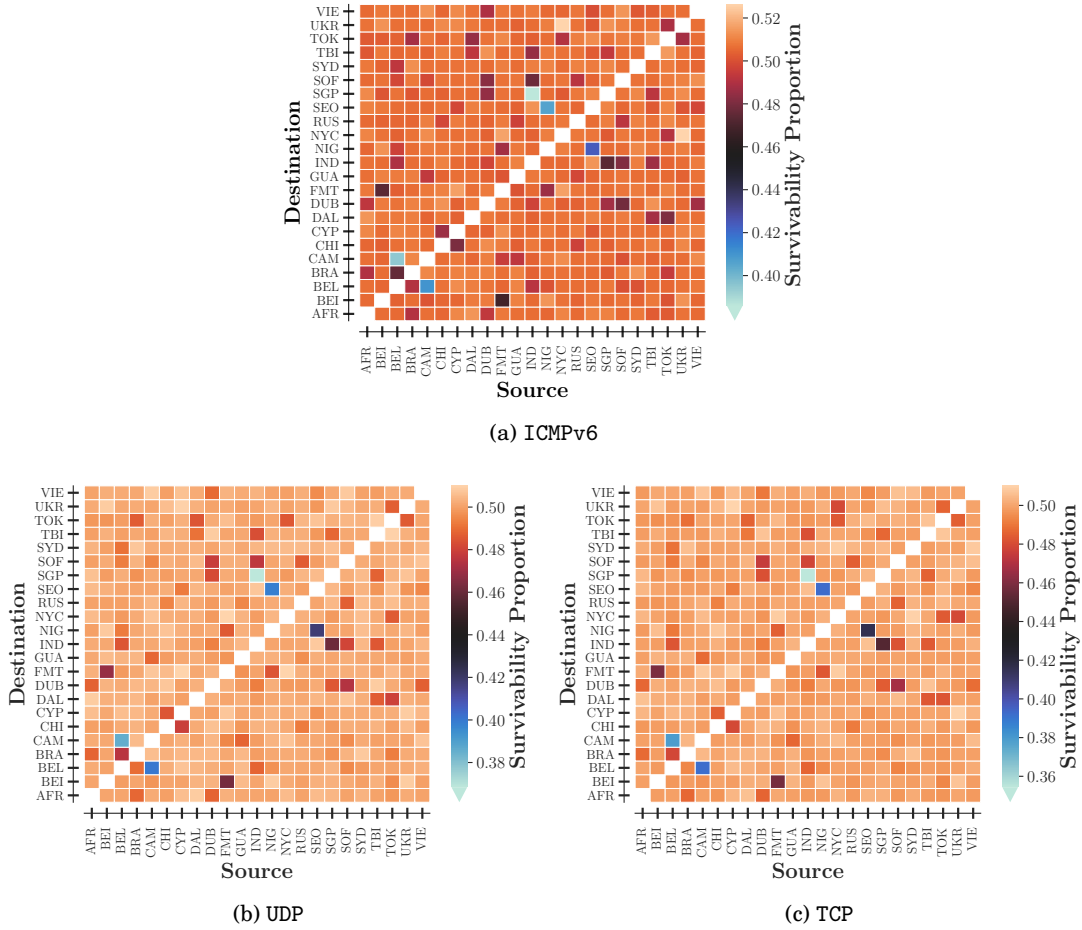


Figure 5.4: High level overview of Extension Headers survivability between CPs, according to a specific upper-layer protocol. Results have been merged between the five Runs.

different Extension Header types and sizes. Tick labels used in Fig. 5.4 refer to labels in Table 5.2. Extension Headers survivability is summarized per upper-layer protocol, *i.e.*, ICMPv6 (Fig. 5.4a), UDP (Fig. 5.4b), and TCP (Fig. 5.4c). A value of 0 in the heatmap colorbar means that, somewhere on the path between two CPs, an IPv6 packet with any Extension Header is dropped every time, *i.e.*, it never reaches the destination. On the contrary, a value of 1 means that such packets always reach the destination. Intermediate values between 0 and 1 reflect heterogeneous behavior along the path, where packet delivery depends on the specific Extension Header type and size, resulting in partial or inconsistent reachability rather than a strictly binary outcome.

We see that Extension Headers with ICMPv6 offers the best survivability (average of 50.4%), while TCP and UDP offer roughly the same survivability rate (average of 49.67% and 49.9% respectively). Such a slight difference might suggest that policies applied by operators are the main cause, since both ICMPv6 and UDP headers share the same length (*i.e.*, 8 bytes) and less filtering is generally applied to ICMPv6 (except for rate limiting). On the other hand, hardware

limitations would make sense too, since routers do not need to parse the upper-layer with ICMPv6, as opposed to with UDP or TCP (*e.g.*, for Layer-4 ports). The lowest survivability, regardless of the upper-layer protocol, is between SGP and IND (in that direction), *i.e.*, 38.59% for ICMPv6, 36.84% for UDP, and 35.43% for TCP. On the contrary, the highest survivability depends on the upper-layer protocol: between UKR and NYC (in both directions – 52.63%) for ICMPv6, NYC to FMT (50.99%) for UDP, and UKR to CYP (51.01%) for TCP.

To summarize, half of the Paris traceroutes containing an Extension Header are dropped along the path. This result is inconsistent with our results in the controlled environment (see Chap. 4), where we observed a perfect survivability rate with default configuration on routers. Also, an effort was made to identify [2] vendors for each encountered router, as a comparison to detect any known hardware limitations. However, only 85 were identified, which only represents 2% of the total, and we cannot therefore rely on it. The distribution was as follows: Juniper (57), Cisco (16), Huawei (12). In the next section, we investigate drops on a per Extension Header basis and we try to understand the cause.

5.2.2 Per Extension Header type

Fig. 5.5 shows the survivability rate on a per Extension Header basis. For each plot, the results have been merged over the five Runs. It is worth mentioning that the observed survivability rates are consistent over all traces, *i.e.*, if the drop of an IPv6 packet with an Extension Header is observed in one trace during Step₃, then it is also observed in the four other traces. However, in a very limited number of cases (between 0.02% and 0.05%), we detected inconsistent experiments in which an IPv6 packet with an Extension Header was dropped in a few traces, and reached the destination in others. Some incomplete vanilla traces (Step₂ and Step₄) were also detected: between 1.05% and 1.32% of two incomplete vanilla traces, and between 0.49% and 0.61% of only one (out of two) complete vanilla trace. More generally, we can say that the above does not affect our results and we believe such situations are most probably due to temporary failures. To be on the safe side, they were excluded from the results.

Fig. 5.5a shows the Routing Header survivability according to its type (first value in the X-Axis couple) and its size (second value in the X-Axis couple – $\emptyset$ for type 2 as it has a fixed and predefined size). It is worth mentioning that a Routing Header mechanism is to be deployed only in limited domains, for security reasons, thus we might see a low survivability. In Fig. 5.5a, we see that the Routing Header survivability is dependent on its size, but not necessarily on its type. In particular, the best survivability (> 70%) is obtained with a 24-byte Routing Header (*i.e.*, the smallest size), while considering larger ones leads to a low survivability (< 10%). There is no real difference between upper-layer protocols, except for a small Routing Header where UDP provides a slightly better survivability. The fact that Type 0 suffers from more drops is not surprising since it is deprecated. This one is definitely due to policies applied by operators, as well as other 24-byte Routing Header types. Indeed, it is highly unlikely that we are hitting

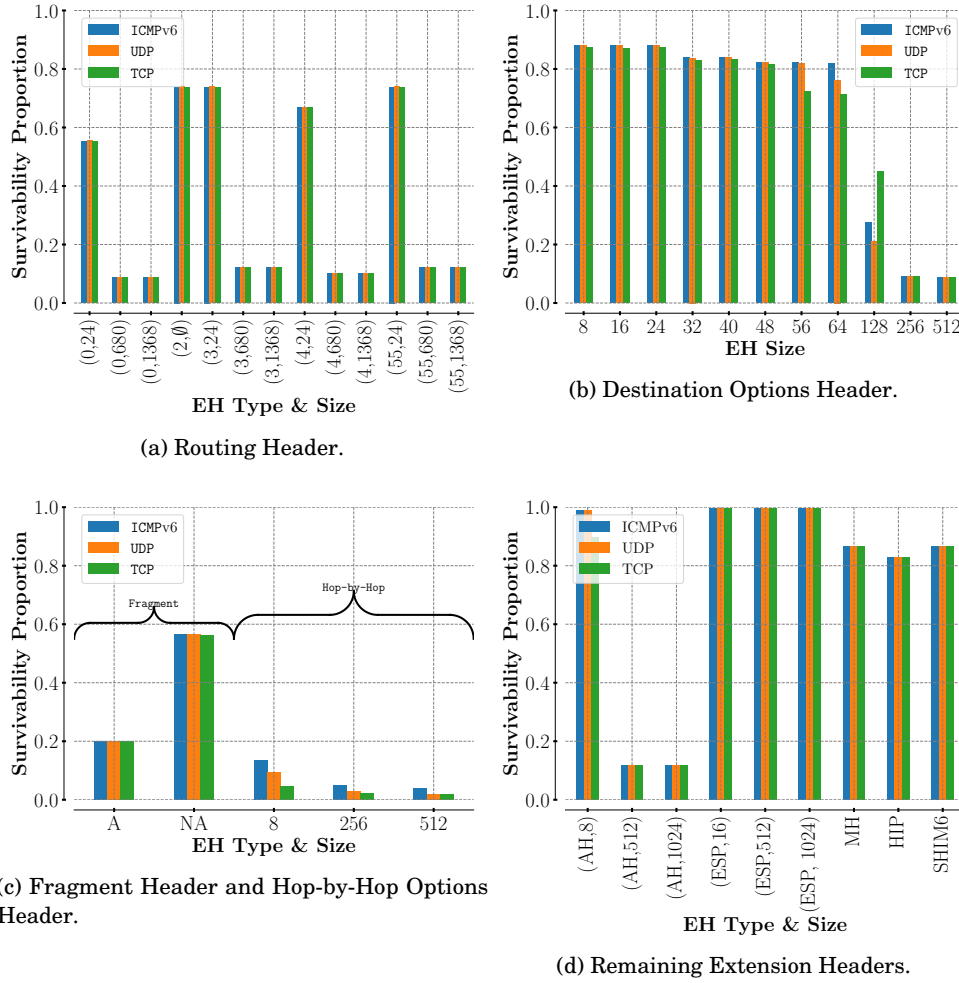


Figure 5.5: Survivability on a per Extension Header basis. Results have been merged over the five Runs.

hardware limitations with only a 24-byte extra in the header. Those policies are applied on Layer-3 and are therefore completely independent of the upper-layer protocol. Types 2 (Mobility support), 3 (RPL), 4 (Segment Routing), and 55 (undefined type) receive the same treatment, except maybe for Segment Routing that suffers from a few more drops, which could be explained by the fact that it is more and more deployed and is therefore treated more aggressively for security reasons (*i.e.*, another good example of policies). As for a larger Routing Header, no one can say for sure by looking at Fig. 5.5a alone if it is due to policies or hardware limitations. It would be tempting to point the finger at policies again but, on the other hand, there is no denying that there seems to be a pattern that might suggest hardware limitations. Indeed, it is not impossible that some traversed ASes still have old routers deployed. Overall, having a low Routing Header survivability on the Internet is not a problem, since the Routing Header should be deployed

in limited domains. The only condition for operators is to ensure that their hardware supports Routing Header deployment.

Fig. 5.5b focuses on the Destination Options Header survivability, according to its size. We expect IPv6 packets with a Destination Options Header to have a high survivability, due to its nature (*i.e.*, processed only by the destination). In Fig. 5.5b, we see that the Destination Options Header is slightly reliable until 64 bytes ($> 70\%$). On the contrary, for larger sizes, the survivability strongly drops below 20%, except for TCP with a 128-byte Destination Options Header which offers a survivability of 45%. Again, as for the Routing Header, there is no strong difference according to the upper-layer protocol for a small Destination Options Header, but ICMPv6 performs slightly better than UDP and TCP. For sizes 56 and 64, TCP performs worse than UDP, which would suggest hardware limitations based on that pattern (*i.e.*, parsing buffer limits are hit more quickly since TCP header is bigger than UDP or ICMPv6), but tends to decrease more slowly than ICMPv6 and UDP until 128 bytes, which would suggest policies instead. The difference with TCP was investigated since middleboxes were suspected to apply different treatment, but only a few MSS modifications were found and were not problematic. Overall, the Destination Options Header can find many use cases on the Internet. However, there is a risk that IPv6 packets will be dropped if they contain a Destination Options Header, especially one larger than 64 bytes.

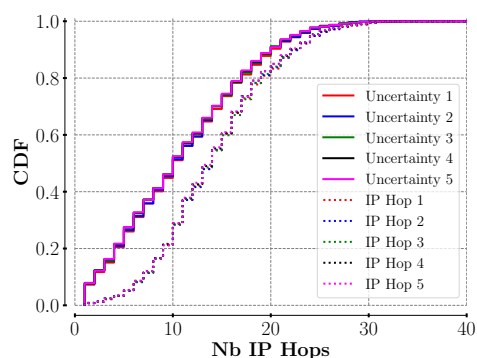
Fig. 5.5c shows the survivability of both a Fragment Header and a Hop-by-Hop Options Header. We focus first on the Fragment Header as an Atomic fragment (“A” on Fig. 5.5c – a packet that contains a Fragment Header without being actually fragmented into multiple pieces, *i.e.*, the fragment offset is 0 and the M (more) bit is also 0) and as a Non-Atomic fragment (“NA” on Fig. 5.5c). We expect a low survivability for Atomic fragments since they are considered harmful [112], but a high survivability for Non-Atomic fragments since this is how IPv6 fragmentation works. Surprisingly, it appears that none of them are reliable, although Non-Atomic fragments survive more easily ($\sim 60\%$) than Atomic fragments (20%). Atomic fragments are more frequently dropped for security reasons [107]. Considering the small fixed size of a Fragment Header (*i.e.*, 8 bytes), hardware limitations might be ruled out. On the other hand, it could be the same phenomenon suggesting hardware limitations, *i.e.*, routers (or security devices reassembling fragments) are not able to find Layer-4 ports (*e.g.*, for filtering or a lookup), except for the first fragment, resulting in a drop. However, if that were the case, the survivability rate for Non-Atomic fragments should be as low as other Extension Headers larger than 64 bytes (*i.e.*, $< 20\%$). But they actually have a survivability rate of almost 60%. Therefore, we believe the main cause is operators’ policies, probably due to security concerns and, to some extent, to performance concerns. Overall, a Non-Atomic fragment is definitely designed for the Internet, but it is hard to rely on it for IPv6 fragmentation, which is bad news. Instead, operators seem to rely on a fixed and conservative MTU value. Note that the Fragment Header might be allowed for some specific services such as large DNS requests [138]. Second, Fig. 5.5c also shows the survivability of

the Hop-by-Hop Options Header (processed by all devices along the path), according to its size. The observed Hop-by-Hop Options Header survivability is quite low ($< 10\%$), which makes it totally unreliable, even with the smallest 8-byte size. Therefore, the main reason is definitely operators' policies. Indeed, the Hop-by-Hop Options Header is heavily dropped for security and performance reasons, regardless of the size. It is not necessarily due to the fact that it goes to the slow path, considering the low survivability rate of even a small 8-byte Hop-by-Hop Options Header. Overall, the Hop-by-Hop Options Header lacks use cases outside of limited domains to make operators change their mind, despite the existence of the Minimum Path MTU Hop-by-Hop Option [130], which still does not seem enough to sacrifice security or performance.

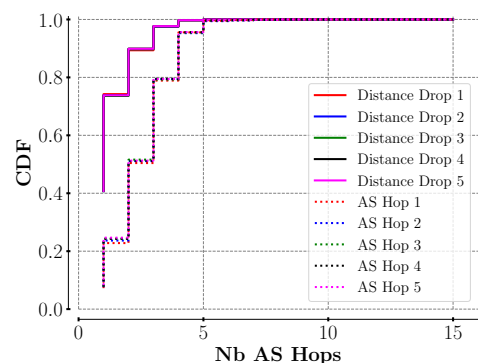
Fig. 5.5d shows the survivability of other Extension Headers. While ESP offers perfect survivability regardless of its size, AH survivability is terrible for large sizes, which reminds us the same pattern observed previously. An 8-byte AH offers near-perfect survivability with ICMPv6 and UDP, and good survivability with TCP ($\sim 90\%$), which is a clear consequence of operators' policies (*i.e.*, ESP is generally preferred and more used for security reasons). In fact, all of this was expected. With ESP, since the payload is encrypted, deep packet inspection is impossible. Therefore, the only filter an operator could come up with would be to forbid ESP, which would not make sense nowadays. Also, should an IPv6 packet with ESP go through a router with hardware limitations (*i.e.*, small parsing buffer), it would do so without problem since the router would not have (*i.e.*, would not be able) to parse deeper than Layer-3. This is another great example of how encryption is important (*e.g.*, to bypass unwanted filters). On the contrary, with AH, the payload is not encrypted, which means it can be treated the same as any other Extension Headers (*i.e.*, probably highlighting hardware limitations). Finally, Mobility Header, Host Identity Protocol Header, and Shim6 Protocol Header offer quite good survivability, without any distinction between upper-layer protocols, even though Mobility Header and Host Identity Protocol Header are not standardized with an upper-layer protocol yet. Considering their respective small sizes (*i.e.*, 8, 48, and 8 bytes), this suggests that they hit operators' policies as well. Note that hardware limitations could also be the reason, highlighted by a slight survivability difference for Host Identity Protocol Header (48 bytes) compared to Mobility Header and Shim6 Protocol Header (both 8 bytes). Overall, these Extension Headers are designed for the public Internet and a perfect survivability would be desirable. However, both operators' policies and hardware limitations seem to prevent it, except for ESP due to its nature.

5.2.3 Drop Attribution

This section tries to identify Autonomous Systems (ASes) which are responsible for dropping IPv6 packets with Extension Headers. The problem is not trivial due to classic traceroute issues (*i.e.*, incomplete traces). Indeed, some hops along the path may not respond to probes during a given traceroute but may respond to subsequent ones. This behavior is usually due to rate limiting [7] or ICMPv6 "Time Exceeded" responses being lost on their way back to the source.



(a) Uncertainty zone size distribution, with respect to traceroute length.



(b) Distance (in terms of AS hop) distribution of Extension Headers drop, with respect to traceroute AS path length.

Figure 5.6: Extension Headers drop attribution.

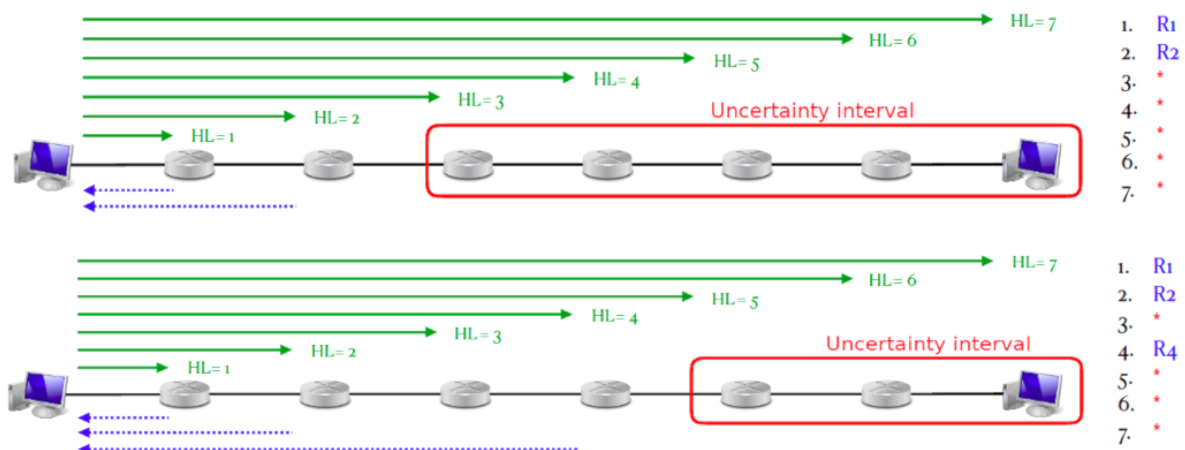


Figure 5.7: Example of uncertainty zones.

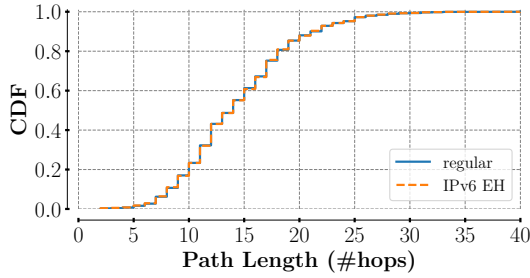
More problematic, some hops never respond, probably because they are configured not to. We also have observed traces with load balancing effects or path modifications (roughly, 8–9% of the cases, on average, for Step₃ for all experiments in a Run), despite our efforts to rely on Paris traceroute to maintain flows. This is discussed in Sec. 5.2.4 and Sec. 5.2.5.

To deal with traceroute issues, we introduce the notion of *uncertainty zone*, *i.e.*, a set of potentially guilty hops, starting with the hop following the last one to respond, and ending at the penultimate hop (*i.e.*, the one before the destination), as illustrated in Fig. 5.7. From that, we derive the uncertainty zone size as the number of hops between those two points. Note that the destination is not part of the guilty possibilities because we have access to it and we made sure all Extension Headers were accepted. Obviously, the earlier a drop in a trace, the bigger the uncertainty zone, and the more difficult it becomes to assign the drop responsibility to an AS,

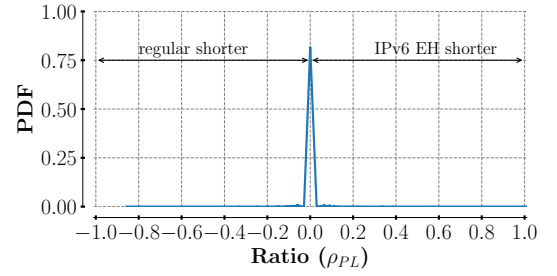
especially when the uncertainty zone spans multiple ASes and when multiple hops never respond. Said otherwise, the uncertainty zone size spans between 1 (the best case, as the uncertainty zone has the smallest size) and the entire path length. Fig. 5.6a compares the uncertainty zone size distribution and the path length distribution (in terms of IP hops). In particular, curves labeled “IP Hop n ” refer to the path length distribution of Run n , while “Uncertainty n ” is the uncertainty zone size distribution for Run n . We first notice that most of the paths are between 5 and 25 hops long. It is not surprising to have longer paths (25+ hops) due to how CPs are spread all over the world. Some anomalies were also detected, *i.e.*, traces with a loop, which explains some extreme hop counts (*e.g.*, 35+). Over the five Runs, we have between 15% and 17% of *successful* vanilla traces (*i.e.*, Step₂ and Step₄) with a loop, and between 8% and 9% of *successful* Extension Headers traces (*i.e.*, Step₃) with a loop. More importantly, we notice that most of the uncertainty zone sizes are between 2 and 20, which confirms that many traces have an uncertainty zone so large that it complicates the responsibility attribution of ASes.

In order to find the AS responsible for a drop, we chose to follow this simple assumption: *an AS will generally apply Extension Headers filtering on ingress* [109]. This assumption and trend has been confirmed by contacting different operators. Of all the operators who responded to our survey and who filter Extension Headers, 100% declare they apply filters on the edges (*i.e.*, on ingress), with 37.5% of them applying filters everywhere (*i.e.*, in the core as well). Note that some ASes may do it on egress as well, like some CPs that prohibit sending Extension Headers out. Based on the aforementioned rule and the fact that we reduce the probability of having errors by doing five Paris traceroutes with Extension Headers (Step₃) for a single experiment, we implemented the following algorithm: (i) we find the AS Number (ASN) for each hop in a trace and rebuild the AS path; (ii) we take the last responding AS in the AS path of each Extension Header trace; (iii) among these, we remember the most distant AS based on the AS path from vanilla traces; (iv) we declare the next AS guilty, or the current one if it is the destination. Fig. 5.6b compares the distance (in terms of AS hops) distribution of ASes responsible for dropping (curves labeled “Distance Drop n ”, where n refers to Run n) and the AS path length of the full trace (labeled “AS Hop n ”). We first notice that most of the paths are between 1 and 5 ASes long. More importantly, we notice that, based on our algorithm, most of the drops seem to occur early in the path, *i.e.*, within 1–3 ASes. It is worth mentioning that the Extension Header drop attribution algorithm was applied to 96.9% of all experiments (consistent over the five Runs). The other 3.1% represents experiments where the algorithm could not find a satisfactory solution. As a result, the drop percentage of Extension Headers per AS type is consistent over the five Runs as well and is as follows: Transit (46.95%), CP (36.03%), Tier1 (13.68%), Stub (3.31%), and IXP (0.03%).

Overall, Extension Headers seem to have more difficulties passing through Transit and CP ASes. It is not surprising for CP ASes, *i.e.*, an Extension Header is likely to hit strict policies, or a bug due to infrastructure complexity, both leading to packet drops. On the other hand, and according to their role, Transit ASes could do a better job and relax their policies regarding



(a) Cumulative distribution of path length (#hops) over the five Runs for both Paris traceroute and Extension Headers.



(b) Path length ratio (ρ_{PL} – See Formula 5.1) between Paris traceroute and Extension Headers.

Figure 5.8: Path length comparison between Paris traceroute and Extension Headers (data from Step₂, Step₃, and Step₄).

Extension Headers, or upgrade their hardware in case of limitations. Also, based on the algorithm naivety and the proximity between CP and Transit ASes in collected traces, it is possible that a part of guilty Transit ASes might be attributed to CP ASes instead. Others do a decent job, even though Tier1 ASes could do better. A good balance must be found between avoiding the ossification of the IPv6 protocol and security. It probably starts with educating people on this topic, so that operators can better evaluate the pros and cons of their policies. Depending on where someone operates, a small decision can have big consequences.

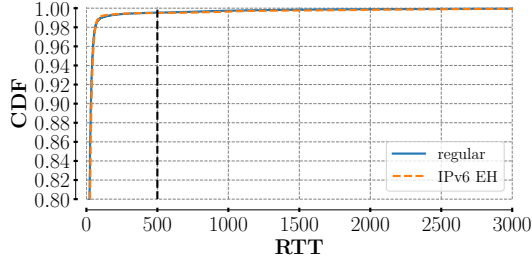
5.2.4 Impact on Path Length

As mentioned previously, we have observed traces with load balancing effects or path modifications (roughly 8–9% of the cases, on average), despite relying on Paris traceroute to maintain flows. Therefore, we want to understand if Extension Headers have an impact on the path length (*i.e.*, number of hops). For each experiment, we compare the two vanilla traces with the five Extension Header traces. Only complete traces with no loop are considered, others are ignored. Fig. 5.8a shows that most of complete traces are approximately between 10 and 20 hops long. There is no difference between the path length of a vanilla trace and the path length of an Extension Header trace, for a same experiment. It means that despite load balancers and path modifications, the number of hops remains consistent.

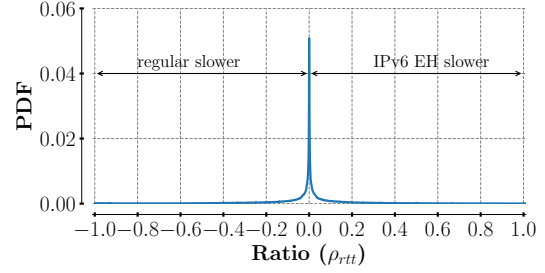
To better observe potential differences in path lengths between vanilla and Extension Header traces, we compute a path length ratio as follow:

$$\rho_{PL} = \frac{\#Hops_{EH} - \#Hops_{regular}}{\#Hops_{regular}}. \quad (5.1)$$

$\rho_{PL} \in [-1, 1]$, where negative values mean that vanilla traces are shorter than Extension Header traces. Positive values, on the contrary, mean that Extension Header traces are shorter.



(a) Cumulative distribution of RTTs over the five Runs for both Paris traceroute and Extension Headers.



(b) RTT ratio (ρ_{RTT} – See Formula 5.2) between Paris traceroute and Extension Headers.

Figure 5.9: Round-trip time (RTT) comparison between Paris traceroute and Extension Headers (data from Step₂, Step₃, and Step₄).

Obviously, a null value means that both traces have the same number of hops. Fig. 5.8b plots ρ_{PL} for the five Runs merged. It shows that the ρ_{PL} distribution is quite symmetric and centered on 0, which confirms that both traces have generally the same length. Overall, Extension Headers do not have an impact on path length (*i.e.*, Extension Headers do not arbitrarily expand or shorten paths).

5.2.5 Impact on Round-Trip Time

This section aims at understanding if Extension Headers have an impact on the round-trip time (RTT). For each experiment, we compare the two vanilla traces with the corresponding five Extension Header traces. Only complete vanilla traces with no loop are considered, as well as any Extension Header traces with no loop, while others are ignored. Fig. 5.9a shows that most of the RTTs are approximately between 0 and 150 milliseconds. There is no difference between the RTTs of a vanilla trace and the RTTs of an Extension Header trace, for a same experiment. It means that routers do not spend any extra time processing Extension Headers. Which makes sense, as the only Extension Header processed by routers in transit is the Hop-by-Hop Options Header. Since the Hop-by-Hop Options Header is very often dropped early on a path (or simply ignored by routers), and given the experimental option code used in the Hop-by-Hop Options Header (*i.e.*, not implemented by routers willing to process the Hop-by-Hop Options Header), we do not see any difference regarding the RTT, although processing a Hop-by-Hop Options Header (whether in slow path or not) does definitely increase RTTs.

In the fashion of path length, and to better understand the potential RTT differences between vanilla and Extension Header traces, we compute an RTT ratio as follow:

$$\rho_{RTT} = \frac{RTT_{EH} - RTT_{regular}}{RTT_{regular}}. \quad (5.2)$$

$\rho_{\text{RTT}} \in [-1, 1]$, where negative values mean that vanilla traces are slower than Extension Header traces. Positive values, on the contrary, mean that Extension Header traces are slower. Obviously, a null value means that both traces follow roughly the same RTT. Fig. 5.9b plots ρ_{RTT} for the five Runs merged. In the fashion of ρ_{PL} , the distribution is symmetrical and centered on 0. Overall, for our datasets, Extension Headers do not have an impact on the RTT, as long as routers do not process them in transit (e.g, Hop-by-Hop Options Header). This also suggests that there were no packet recirculation in routers with dedicated hardware, or very little at least, as this would increase RTTs.

5.2.6 Discussion

To summarize this chapter, it appears that size matters when it comes to Extension Headers [109]. The larger they are, the more likely to get dropped. Whether this is due to policies or hardware limitations is debatable on a case-by-case basis but, overall, both are observed causes. While policies are probably more frequent, it only takes one old router with hardware limitations on a path to become the cause of the drop. Note that Extension Headers modified on transit were also considered. However, 100% of the Extension Headers received were unmodified. Now, let us compare all these observations with the responses received to our survey. This comparison with the survey results is to be taken with a pinch of salt, because none of the respondent ASes were traversed during our measurements, and because it only represents a small set of operators. First, 81.8% of operators declare they do not explicitly filter Extension Headers, while the other 18.2% declare they do. This could indeed be representative of what we observed. Should it be the case, this could highlight hardware limitations as the main, most frequent, cause of drops. A good illustration is the following comment from an operator: *“While we do not intentionally filter Extension Headers, we do not intentionally use them either and have never done any testing to confirm they are functional on our network”*. Also, those who explicitly filter Extension Headers do it mainly on the type (100%), while filters on the size (12.5%) and on the content (12.5%) seem less frequent. Based on what we observed, it is representative for the type but not necessarily for the size. The latter could again highlight hardware limitations. Finally, 72.7% of respondents declare they apply a filter on Layer-3 (e.g, next header, source or destination address), while 61.4% declare they apply a filter on Layer-4 (e.g, ports). While it was proved in Chap. 4 that it does not cause issues with Extension Headers on recent routers, trying to filter on the IPv6 “Next Header” field versus the *protocol* (i.e, Layer-4 “proto”, the upper-layer protocol after the Extension Headers chain) is not the same. Indeed, should an operator intentionally use (or not) the former and only accept, e.g, TCP, UDP, and ICMPv6, any Extension Header would therefore not be allowed. After all, it is up to operators to decide what they accept or not. Extension Headers are controversial and some operators would strictly forbid them, as illustrated by the following comments received to our survey: “death to extension headers”, as well as “Extension Headers were a trash idea and need to be eradicated”. Some others are more moderate and listen to IETF

discussions, as illustrated by the following comment: “We are following the current tests and waiting for BCP in relation to Extension Headers processing”. This one shows that good RFCs and BCPs could influence operators’ decisions on Extension Headers.

SUMMARY

The first part of this document aimed to answer the first Research Question (**RQ-1**) regarding IPv6 Extension Headers, namely “*Can we actually use IPv6 Extension Headers on the Internet?*”. This chapter summarizes the points covered so far and, based on the data collected, concludes by answering **RQ-1**.

In Chap. 1, we provide a detailed overview of what Extension Headers are, how they work, and cover an exhaustive list of types and options. In Chap. 2, we review the related work, both in research and standardization. In Chap. 3, we introduce a new eBPF-based tool, called FISHNET, to easily inject Extension Headers in IPv6 traffic. This tool is used for measurements in Chap. 4, where we investigate whether IPv6 packets with Extension Headers pass through routers from different vendors with default configurations. Finally, in Chap. 5, we perform large-scale measurements on the Internet with FISHNET, provide an overview of how Extension Headers are processed, and compare the results with observations made in Chap. 4.

Now, based on what we learned, let us conclude by answering **RQ-1**: “*Can we actually use IPv6 Extension Headers on the Internet?*”. In other words, can anyone use IPv6 Extension Headers, from anywhere to any location, and for any use case? The answer is “*it depends*”. First, several studies [9, 59, 74, 75, 110, 139, 140, 193] have reported different results regarding how Extension Headers survive over the Internet, simply because they measured from different locations (*i.e.*, Internet Core vs. Edge vs. Access Networks, inbound vs. outbound). For example, inbound IPv6 traffic with Extension Headers to Access Networks is likely to be dropped by service providers, some cloud providers have strict Extension Headers filtering (usually both inbound and outbound), etc. Then, it depends on the Extension Header type itself. For example, a Hop-by-Hop Options Header has a very low probability of survival in general (< 15%), compared to a Destination Options Header that has a better chance (~80%). Finally, it also depends on the Extension

Header size. It appears that the survival probability is ~80% for up to 64 bytes of Extension Headers, except for a Hop-by-Hop Options Header (< 15%) or a Fragment Header (< 60%). Overall, operators should first test Extension Headers survivability based on their needs (*e.g.*, with a tool like FISHNET).

On the other hand, some Extension Headers were not originally designed for the public Internet and are deployed in limited domains instead (*e.g.*, the Routing Header). Therefore, it is not really an issue if those do not survive well on the public Internet. In limited domains, all Extension Headers can potentially be used by operators anyway, as long as there is hardware support (*e.g.*, available implementations, sufficiently large parsing buffers, etc). However, there are at least three types of Extension Headers that should be deployable on the public Internet: the Fragment Header, the Destination Options Header, and to some extent the Hop-by-Hop Options Header. The latter could find useful use cases (*e.g.*, with the Minimum Path MTU Hop-by-Hop Option [130]), but someone cannot expect others to perform additional processing on their routers without prior agreement (for obvious reasons of performance, security, and cost). The best trade-off would be for routers to ignore the Hop-by-Hop Options Header instead of dropping IPv6 packets that contain one, as specified in RFC 9673 [131]. Regarding IPv6 fragmentation, it is broken in general because the Fragment Header (only 8 bytes) is not well supported on the public Internet, and people tend to rely on a fixed and conservative MTU value instead. As for the Destination Options Header, it is useful for edge-to-edge use cases (*e.g.*, telemetry) on the public Internet. There is still hope, at least if its size does not exceed 64 bytes. To bypass operators' policies and avoid hardware limitations, one possible approach is to use the Destination Options Header with ESP. As a result, the Destination Options Header would survive 100% of the time regardless of its size. In principle, the same technique could be applied to other Extension Headers in order to "hide" them. However, this approach is not compliant with the order of Extension Headers specified in RFC 8200. Although the specification does not use normative language, meaning that some vendors may permit such behavior, the use of ESP would prevent transit nodes from accessing Extension Headers that appear after it and therefore from participating in network operations.

Finally, let us discuss operators' policies and hardware limitations one last time. Against the former, there is not much we can do, except perhaps try to educate people about Extension Headers and hope for the best. Indeed, even sometimes IETF RFCs or BCPs are not followed. However, against hardware limitations, standards bodies (*i.e.*, the IETF) can do something in the long term. A recent example is the IETF draft [128] titled "*Limits on Sending and Processing IPv6 Extension Headers*". Its goal is to specify (*i.e.*, report, document) a minimum size limit of 64 bytes for Extension Headers, making them usable on the public Internet up to that size. While the motivation behind the draft makes sense, the proposed solution is controversial and has received pushbacks [84]. Indeed, in the "best" case scenario, the draft would be ignored by the industry and it would not change anything. In the worst case, hardware vendors would interpret

this lower limit as an upper limit, thus locking Extension Headers for good and rendering them unusable in the future. Of course, whatever the adopted solution, we would still need to wait for the end of life of all old hardware with limitations. Therefore, it should improve over time, as surprising as it may be, as long as operators do not make the situation worse with their policies. Sometimes perfect is the enemy of good, and the best we can do right now is probably to wait, instead of complicating things even further. Nevertheless, a possible action plan for the IETF is discussed in the future work section (Chap. 13), which avoids defining limits on the size of Extension Headers.

Since Extension Headers can be used in limited domains, the second part of this document studies their use to improve network observability.

PART

III

ADVANCED NETWORK OB- SERVABILITY

BACKGROUND

The Internet has evolved a lot since the early researches in topology discovery [69]. Initially as a hierarchical and sparsely interconnected network, it became a flatter and much more densely interconnected network [66, 103, 304] in which hypergiant distribution networks (*e.g.*, Akamai, Amazon, Cloudflare, Google, Meta, or Netflix), also called hypergiants, play a crucial role. Nowadays, such large and globally distributed Content Delivery Networks (CDNs) operated by major technology companies are responsible for a significant part of the total Internet traffic due to their ever-growing number of users [25]. In order to handle all the traffic while providing customers with an optimal user experience (*i.e.*, low latency), these companies have been deploying interconnected infrastructures all around the world across many data centers relying on Data Center Networks (DCNs) [43].

Modern networks demand accurate, timely, and scalable insight into their operational status. With increasing complexity arising from virtualization, encrypted transport protocols, and highly dynamic topologies, traditional monitoring tools are sometimes no longer sufficient to capture the full spectrum of network behavior. Over the years, several Operations, Administration, and Maintenance (OAM) tools have been developed for various layers of the protocol stack [213]. The measurement techniques developed under the OAM framework can perform fault detection and isolation, and provide performance measurements. Therefore, data collection (*i.e.*, telemetry – see definition below) is not only critical for hypergiants, but also for DCNs and network operators in general, in order to address two particular challenges [183, 302]. First, network infrastructures must guarantee high availability, ensuring continuous operation, even in case of a failure. Second, the service must be provided with the highest quality and minimal delay. Consequently, operators must carefully engineer their network to maximize the Quality-of-Experience (QoE) for their users.

Table 7.1: Examples of Metrics, Events, Logs, and Traces (MELT): Application vs. Network telemetry.

	Application telemetry	Network telemetry
Metrics	CPU usage, request latency, error rates	Packet loss, jitter
Events	Deployment, service restart	Link state, route change
Logs	Application logs, stack traces	Syslogs
Traces	Request to a database	Flow-level traces

This chapter provides the definition of *telemetry* (Sec. 7.1), which is the required foundation to understand what monitoring and observability are (Sec. 7.2). Then, the main OAM protocols and tools to measure and observe networks are discussed (Sec. 7.3).

7.1 Telemetry

The Cambridge Dictionary has the following definition for the word **telemetry**: “*the science or process of collecting information about objects that are far away and sending the information somewhere electronically*” [39]. In other words, this is the technical term to describe the collection and transmission of measured data from remote devices, forming the foundation for both monitoring and observability (Sec. 7.2).

There exist two main types of telemetry, namely application telemetry (Sec. 7.1.1) and network telemetry (Sec. 7.1.2). They focus on different levels of the network stack and, although they can coexist quite easily, it is difficult to combine (*i.e.*, correlate) them. This challenge is discussed and addressed in Chap. 12.

The fundamental telemetry data types are represented by the MELT (*i.e.*, Metrics, Events, Logs, and Traces) model:

- **Metrics:** Numerical data that provides insights into system performance.
- **Events:** Timestamped records of occurrences that happened in a system.
- **Logs:** Timestamped records that capture detailed context or diagnostic information about a system.
- **Traces:** Time-ordered records that capture the end-to-end request flow through a system.

Although the concept of traces (*i.e.*, the “T” in MELT) is rare in network telemetry, it is theoretically possible. However, traces generally find their scope in application telemetry. Table 7.1 provides examples for each MELT component in both contexts (*i.e.*, application and network telemetry).

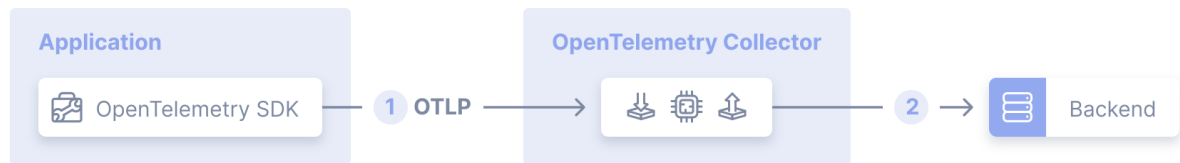


Figure 7.1: Overview of OpenTelemetry. [229]

7.1.1 Application Telemetry

Application telemetry is an application-level measurement, typically in Layer-7, with indirect visibility into Layer-5 and Layer-6. It usually requires to add code or instrumentation to an application to collect and transmit telemetry data. It provides insights into performance and behavior of applications. To achieve application telemetry, two main solutions dominate the landscape: OpenTelemetry (OTel) [231, 232] and Application Performance Management/Monitoring (APM) tools.

OpenTelemetry is a free open-source collection of APIs (Application Programming Interfaces), SDKs (Software Development Kits), and tools for generating, collecting, and exporting telemetry data (*i.e.*, MELT) to a variety of analysis tools (*i.e.*, backend systems). It defines vendor-neutral standards and language-agnostic definitions for applications to produce telemetry, therefore ensuring interoperability across vendors and programming languages. OpenTelemetry integrates with many popular libraries and frameworks, and supports code-based and zero-code instrumentation. A key component of OpenTelemetry is the collector, which is a service that receives, processes, and exports telemetry data. The OpenTelemetry collector includes many exporters to be compatible with existing backend systems. As illustrated in Fig. 7.1, an application is instrumented with the OpenTelemetry SDK to send telemetry signals to an OpenTelemetry collector using the OpenTelemetry protocol (OTLP). Then, the OpenTelemetry collector exports telemetry data to one or more backend systems. In summary, OpenTelemetry is not a backend system, *i.e.*, it does not provide visualization, alerting, or analytics capabilities. It provides application instrumentation and telemetry data collection. Therefore, OpenTelemetry must be used with an APM tool or a backend system.

Application Performance Management/Monitoring (APM) is a set of tools and services for tracking, analyzing, and improving the performance and availability of applications. APM tools provide visualization and alerting mechanisms, as well as built-in dashboards. Most of APM tools are commercial (*i.e.*, paid solutions) and can run independently. They have their own APIs, SDKs, and tools (*e.g.*, collectors), and therefore do not require OpenTelemetry (or anything similar) to run. However, many vendors now support OpenTelemetry standards, giving customers the possibility to send telemetry data collected with OpenTelemetry into their platform. The most famous are Dynatrace [71], Datadog [61], New Relic [224], Splunk [263], Elastic [73], and Instana [146]. There are also several free open-source APM tools such as Apache SkyWalking [266], SigNoz [256],

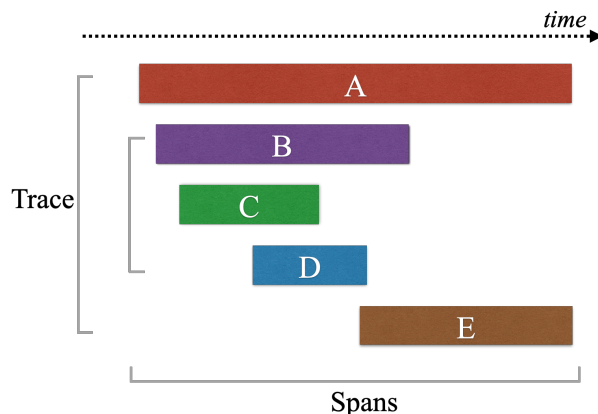


Figure 7.2: Relation between traces and spans in distributed tracing.

and Jaeger [167], with some of them supporting OpenTelemetry standards. Does it mean that commercial APM tools are in danger? Not really, but vendors are now facing more price pressure and competition on features. Although they are paid solutions, the main remaining advantages are the ease of deployment, the use of advanced analytics, the enterprise support, the integration ecosystem, and the mature user experience.

This document, and more specifically Chap. 12, will rely on the combination of OpenTelemetry and Jaeger, which makes it a completely free and open-source solution. Jaeger is often mentioned in the context of APM tools. However, it is in fact a distributed tracing tool, originally developed at Uber [284] and now a Cloud Native Computing Foundation (CNCF) project [271, 272]. Unlike full APM solutions, Jaeger does not natively handle metrics or logs, but instead focuses exclusively on tracing (*e.g.*, provided by OpenTelemetry in this case), which is an important subset of APM tools. Jaeger can be combined with complementary backends, respectively for metrics (*e.g.*, Prometheus [243]) and logs (*e.g.*, Loki [117]), to form a complete APM solution. Distributed tracing is based on two fundamental concepts: traces and spans. OpenTelemetry has the following generic definitions [230]. A **span** “represents a unit of work or operation. Spans track specific operations that a request makes, painting a picture of what happened during the time in which that operation was executed”. A **trace** “is made of one or more spans and records the paths taken by requests (made by an application or end-user) as they propagate through multi-service architectures, like microservice and serverless applications”. Fig. 7.2 illustrates the concept of traces and spans in distributed tracing. Spans can represent any operation to be observed in applications. The trace represents an entire request, *i.e.*, all operations (spans A, B, C, D, and E). In this example trace, span A represents a specific operation (*e.g.*, user login) that triggers two sub-operations (*i.e.*, spans B and E), each representing another specific operation (*e.g.*, a query to the database for span B, and the execution of a local algorithm for span E). Spans C and D are two sub-operations of span B (*e.g.*, a fetch and an update). In summary, a trace always represents the whole request, while a span always represents one operation within that trace.

Table 7.2: Difference in MELT coverage between monitoring and observability (network vs. application). An empty-cell means not supported, “~” means sometimes supported, and ✓ means supported.

	Scope	MELT			
		Metrics	Events	Logs	Traces
Monitoring	Network	✓	✓	~	
	Application	✓	✓	~	
Observability	Network	✓	✓	✓	~
	Application	✓	✓	✓	✓

7.1.2 Network Telemetry

Network telemetry is a network-level measurement, typically with visibility into Layer-2, Layer-3, and Layer-4. It provides insights into network performance and health. Telemetry data (*e.g.*, traffic volume, latency, packet loss, queue depth, buffer occupancy, etc) is collected from various network devices, such as switches, routers, and servers.

Sec. 7.3 discusses the OAM protocols and tools for network telemetry.

7.2 Monitoring vs. Observability

Monitoring and observability are key concepts that are often confused with each other. Both rely on telemetry data as a foundation (*i.e.*, data collected *and* exported, ready to be ingested). Although related, they have different meanings and understanding the distinction is essential. This section provides definitions to clarify matters and highlights their difference, particularly with regard to their respective MELT coverage.

The Cambridge Dictionary has the following definition for **monitoring**: “*to watch and check a situation carefully for a period of time in order to discover something about it*” [37]. In the context of networks and systems, monitoring refers to the process of collecting, aggregating, and analyzing predefined telemetry data (*i.e.*, typically metrics and events, and in some cases logs, as shown in Table 7.2) to assess the current state, detect anomalies, and ensure the system is functioning as expected.

The word “*observability*” does not officially exist in dictionaries. However, its etymology [295] is the combination of two words (*i.e.*, *observe* and *ability*), meaning the ability to observe. The Cambridge Dictionary has the following definition for the ability to observe (*i.e.*, **observability**): “*the ability to watch carefully the way something happens or the way someone does something, especially in order to learn more about it*” [38]. In the context of networks and systems, observability is the ability of a system to make its internal state understandable through its telemetry data. In practice, observability goes beyond monitoring by covering the full MELT spectrum (*i.e.*,

Table 7.3: Non-exhaustive list of well-known monitoring tools, with their MELT coverage and scope (network vs. application). An empty-cell means no support, “~” means limited support, and ✓ means full support. Based on native features only (no add-ons, plugins, or external modules).

Monitoring tool	MELT coverage				Scope
	Metrics	Events	Logs	Traces	
Nagios [220]		✓			Network and Application
Zabbix [303]	✓	✓	~		Network and Application
Icinga [147]		✓			Network and Application
Netdata [223]	✓	✓	~		Network and Application
SolarWinds NPM [259]	✓	✓			Network
PRTG Network Monitor [235]	✓	✓			Network
Centreon [42]	~	✓			Network and Application

metrics, events, logs, and traces, as shown in Table 7.2), thereby enabling root cause analysis and proactive troubleshooting.

In summary, monitoring and observability are closely related but distinct concepts. Monitoring can be seen as a subset of observability: it focuses on collecting predefined telemetry data to answer questions such as “*what happened?*” and “*when did it happen?*”. Observability, on the other hand, offers a detailed view of a system to help answer questions like “*why did it happen?*” and “*how did it happen?*”. While monitoring provides the foundation, observability leverages it. A well-monitored system may still be unobservable if it lacks sufficient instrumentation or visibility to answer new and unexpected questions. In contrast, an observable system inherently enables monitoring, since monitoring is a core component of observability, illustrating that observability goes beyond what monitoring alone can provide. This conceptual distinction has also influenced operational practice. Traditionally, networking has primarily focused on monitoring using well-defined metrics to track system health, while applications have embraced observability. However, recent advances are extending observability concepts into the network, thus gradually blurring this distinction.

7.2.1 Monitoring tools

As previously illustrated in Table 7.2, a monitoring tool can be considered as such if it supports at least metrics and, usually, events. Table 7.3 shows a non-exhaustive list of well-known monitoring tools, with their MELT coverage and scope (network vs. application). The classification is based on native features only (no add-ons, plugins, or external modules). There are two commercial tools (*i.e.*, paid): SolarWinds NPM and PRTG Network Monitor. Others are free and open-source. Nagios [220], Icinga [147], and Centreon [42] do not fully support metrics *natively* (*i.e.*, it requires the use of a plugin), mainly due to historical reasons. However, they are still considered monitoring tools. Several tools have evolved a lot over time, which generally tends to blur even more the difference between monitoring and observability tools. For example, Netdata

Table 7.4: Non-exhaustive list of well-known observability tools, with their MELT coverage and scope (network vs. application). An empty-cell means no support, and ✓ means full support. Based on native features only (no add-ons, plugins, or external modules).

Observability tool	MELT coverage				Scope
	Metrics	Events	Logs	Traces	
Prometheus [243]	✓				Application
Mimir [118]	✓				Application
Graylog [120]			✓		Application
Loki [117]			✓		Application
Jaeger [167]				✓	Application
Zipkin [233]				✓	Application
Tempo [119]				✓	Application
Datadog [61]	✓	✓	✓	✓	Network and Application
Dynatrace [71]	✓	✓	✓	✓	Network and Application
New Relic [224]	✓	✓	✓	✓	Application
Splunk [263]	✓	✓	✓	✓	Network and Application
Elastic [73]	✓	✓	✓	✓	Network and Application
Instana [146]	✓	✓	✓	✓	Network and Application
Apache SkyWalking [266]	✓	✓	✓	✓	Application
SigNoz [256]	✓	✓	✓	✓	Application

could be considered an observability tool, but it is more a monitoring tool overall (*i.e.*, advanced monitoring). On the other hand, some observability tools could be considered monitoring tools as well (*e.g.*, Prometheus [243]). It is also worth mentioning Wireshark [298], a famous network monitoring tool. Although not listed here, it is still a monitoring tool.

7.2.2 Observability tools

As previously illustrated in Table 7.2, a full observability tool (*i.e.*, also called observability platform) can be considered as such if it supports metrics, events, logs, and traces. Said otherwise, it can provide full observability by itself, alone. However, an observability tool in general can cover any subset of MELT. For example, in Table 7.4, both Prometheus [243] and Mimir [118] only support metrics, both Graylog [120] and Loki [117] only support logs, while Jaeger [167], Zipkin [233], and Tempo [119] only support traces. They are all free and open-source. But, such observability tools alone do not provide full observability. For that, they must be combined to form a full observability stack. In practice, there is no single “perfect” open-source observability stack (*i.e.*, each use case is different and it depends on the context). As an example, one popular and robust combination is Prometheus (metrics), Loki (logs), Jaeger (traces), Grafana (visualization), and OpenTelemetry (data collection via its standard). One could also replace Jaeger by Tempo in order to stay in the Grafana world. In other words, there are many possible (and efficient) combinations. On the other hand, there are full observability tools already, without the need to

combine a set of tools. Most of them are commercial (*i.e.*, paid), *e.g.*, Datadog [61], Dynatrace [71], or New Relic [224], while few of them are free and open-source, *e.g.*, Apache SkyWalking [266] or SigNoz [256]. Pros and cons of paid tools were previously discussed in Sec. 7.1.1.

7.3 OAM Protocols and Tools

Application-level OAM is provided by APM tools, which was discussed in Sec. 7.1.1. This section focuses on network-level OAM, which represents most OAM protocols.

When discussing OAM in practice, it is important to distinguish between OAM protocols and tools. According to RFC 7276 [213], OAM protocols are standardized mechanisms that define how measurement, fault management, or performance monitoring is achieved. OAM tools, on the other hand, are implementations or applications that make use of these OAM protocols to carry out specific tasks. In other words, OAM protocols provide the foundation, while OAM tools provide the operational means by which network operators and researchers interact with those protocols. In some cases, an OAM protocol can be an OAM tool.

Different networking planes (*i.e.*, data, control, and management) are actively instrumented and monitored by OAM mechanisms. The data plane, or forwarding plane, is responsible for moving packets through the network device. The control plane manages routing and signaling to determine how packets should be forwarded. The management plane is responsible for the configuration and operations of network devices by facilitating the exchange of information. This document focuses exclusively on the data plane, with particular emphasis on data plane telemetry (*i.e.*, telemetry data collected directly from packets).

OAM measurement methods fall into three categories: active, passive, and hybrid. Secs. 7.3.1, 7.3.2, and 7.3.3 provide a definition [144, 217] and list respectively active, passive, and hybrid measurement protocols and tools. These lists are restricted to OAM in the data plane and for the IP layer (*i.e.*, Layer-3, more specifically IPv6) due to the scope of this document, and to standard OAM protocols.

7.3.1 Active Measurement

Active measurements generate and inject dedicated (*i.e.*, “synthetic”) probe traffic into the network to infer performance metrics.

Ping [79, 166, 182] is an OAM tool for testing the reachability of destinations and measuring RTTs. It operates by sending an ICMPv6 Echo Request message and waiting for the corresponding Echo Reply message. Variants such as UDP ping and TCP ping use UDP or TCP instead, but serve the same purpose.

Traceroute [79, 182] is an OAM tool for discovering a path between a source and a destination. It sends a sequence of UDP packets to the destination, starting with Hop-Limit = 1 and increasing until the destination is reached. It uses Hop-Limit expiration and ICMPv6 Time Exceeded

messages to identify each hop. Extensions such as Paris Traceroute [12] improve accuracy in load-balanced environments. There are other implementations that use ICMPv6 or TCP.

Bidirectional Forwarding Detection (BFD) [175, 176] is an OAM protocol for quick fault detection in the path between two forwarding engines. It works by exchanging control messages at a configurable interval, enabling failure detection within milliseconds. It is independent of the underlying link technology, data traffic, and routing protocols (*e.g.*, OSPF [53], IS-IS [36], BGP [246]), and can be used to trigger fast rerouting and improve resilience.

One-Way Active Measurement Protocol (OWAMP) [253] is a framework for measuring one-way characteristics of IP networks, such as packet loss [6] and delay [5]. It requires the clocks of the two endpoints to be synchronized, and includes two separate protocols: OWAMP-Control (*i.e.*, initiate, start and stop test sessions and fetch their results) which runs over TCP, and OWAMP-Test (*i.e.*, exchange test packets between two measurement nodes) which runs over UDP. The latter is the active OAM protocol.

Two-Way Active Measurement Protocol (TWAMP) [123] is a framework for measuring both one-way and two-way (*i.e.*, round-trip) characteristics of IP networks. Like OWAMP, it includes separate TWAMP-Control and TWAMP-Test protocols, with the latter being the active OAM protocol.

Simple Two-way Active Measurement Protocol (STAMP) [208] is a framework for measuring both one-way and two-way (*i.e.*, round-trip) performance metrics, including delay, delay variation (jitter [65]), and packet loss. It provides TWAMP-like capabilities without requiring a control protocol or synchronization. The active OAM protocol is STAMP-Test, which runs over UDP.

7.3.2 Passive Measurement

Passive measurements observe and analyze existing network traffic without injecting additional packets.

Tcpdump [278] is an OAM tool for capturing and recording network traffic (*i.e.*, with libpcap [279]) that traverses an interface. It allows operators to observe traffic for, *e.g.*, diagnosing connectivity issues, verifying protocol behavior, and analyzing packet flows. Another similar and widely used OAM tool is Wireshark [298], which, unlike tcpdump, provides a Graphical User Interface (GUI).

IP Flow Information Export (IPFIX) [51, 52] is an OAM and telemetry protocol for exporting flow-based information from network devices to a collector. IPFIX observes and encodes traffic flows using flexible templates, including standard and vendor-specific information elements (IEs). While the export generates traffic, the collection of flows is passive. IPFIX is commonly used for, *e.g.*, traffic monitoring, performance analysis, and anomaly detection in operational networks.

sFlow [238] is similar to IPFIX. However, instead of exporting flow-based information, it exports sampled packets and interface counters to a collector. sFlow is highly scalable, lightweight,

and is suitable for the same use cases as IPFIX but is optimized for high-speed or large-scale networks.

Chap. 3 already discussed eBPF. It can be considered a passive OAM tool for network monitoring when it does not modify packets, but eBPF can also function as an active or hybrid OAM tool if it generates packets or inserts data into them.

7.3.3 Hybrid Measurement

Hybrid measurements combine both approaches (*i.e.*, active and passive methods) by embedding measurement information into network traffic.

Alternate Marking [94, 95] is an OAM technique for measuring packet loss, delay, and jitter, by marking packets using color bits. This technique requires coordination between endpoints, and can be considered passive or hybrid depending on the application. For example, in IPv6 Alternate Marking [98], ingress nodes that insert marking bits using the AltMark Option in either a Hop-by-Hop Options Header or Destination Options Header are hybrid, and transit nodes that only observe the marks for measurement are passive. Although Alternate Marking is defined for point-to-point unicast flows, it can be extended to any kind of multipoint-to-multipoint paths (*e.g.*, unicast, anycast, and ECMP flows) with Multipoint Alternate Marking [96, 97].

In-band Network Telemetry (INT) [184, 277] is an OAM technique for collecting network performance and path information by embedding telemetry data (*e.g.*, per-hop latency, queue occupancy, packet loss) directly into packets as they traverse the network. While INT is not an IETF standard, it is specified by the P4 (*i.e.*, Programming Protocol-independent Packet Processors, a programming language maintained by the P4 Language Consortium) Applications Working Group. INT requires a programmable data plane and can only be deployed on P4-compatible devices such as programmable switches, routers, or SmartNICs. INT is not specified to be carried directly in the IPv6 header chain. Instead, it can be encapsulated in, for example, transport protocols such as TCP or UDP.

In Situ Operations, Administration, and Maintenance (IOAM) [31] is an OAM technique and an IETF standard for collecting operational and telemetry data, similar to INT, by embedding it directly into packets as they traverse the network. IOAM supports multiple data plane options, such as IPv6 using either a Hop-by-Hop Options Header or Destination Options Header. IOAM can be implemented in a variety of environments, including switches, routers, and host stacks. Like INT, IOAM enables use cases such as performance monitoring, fault isolation, and path verification. Chap. 8 discusses IOAM in detail.

7.4 Pushing further with Advanced Observability

This document aims to improve the state of the art in observability. Advanced observability may be defined as the capability to gain deep, context-rich, and actionable insights into a system's

internal state by extending traditional observability with finer-grained, timely, and correlated telemetry. This document investigates the use of a promising hybrid OAM measurement method (*i.e.*, IOAM) with IPv6 to improve traditional observability. This approach naturally extends to networks, where richer “in situ” telemetry techniques have been developed to complement active and passive measurements. IOAM enables packets to carry telemetry data as they traverse the network, thanks to an IOAM Option in either a Hop-by-Hop Options Header or Destination Options Header, thereby providing unprecedented data path visibility. Beyond network-level observability, IOAM is also investigated as a means to enrich application-level observability, enabling correlation of network and application telemetry. By combining insights from both layers, advanced observability can provide a full picture of how a system behaves.

The following chapters present IOAM in detail, discuss its key use cases, and explore it as a case study of advanced observability in practice: first its implementation in the Linux kernel, followed by extensive performance evaluation, then its integrity protection mechanism, and finally an investigation of network and application telemetry correlation for advanced observability.

IN SITU OPERATIONS, ADMINISTRATION, AND MAINTENANCE (IOAM)

In Situ Operations, Administration, and Maintenance (IOAM) [31] is a hybrid OAM measurement method and an IETF standard (*i.e.*, RFC 9197) designed to collect operational and telemetry information in a network. The term “In Situ” reflects the key idea that OAM data is embedded into data packets as they traverse a path between two points, rather than relying on dedicated OAM packets for measurement. By enabling the collection of per-hop and end-to-end information, IOAM provides detailed visibility into network behavior and performance, as well as potential anomalies. This capability is particularly valuable for modern, high-speed, and dynamic networks, such as data center fabrics and large-scale service provider infrastructures, where traditional out-of-band measurement methods may be insufficient or inefficient for providing timely and accurate information. In addition, IOAM enables advanced use cases such as traffic SLA (Service Level Agreement) verification, service and path validation, and fault detection and isolation. This chapter presents a comprehensive overview of IOAM, covering its architecture, operational principles, deployment considerations, practical use cases, and how it compares to other hybrid OAM measurement methods.

8.1 Terminology

This section introduces key IOAM terms, as defined in RFC 9197 and related IETF specifications, which represent the foundation needed to understand IOAM and how it is deployed within a network. The notions of IOAM-Domains and IOAM-Namespaces are explained, along with the different roles of IOAM nodes (*i.e.*, *encapsulating*, *transit*, and *decapsulating* nodes), to provide a clear understanding of their purpose and relationship.

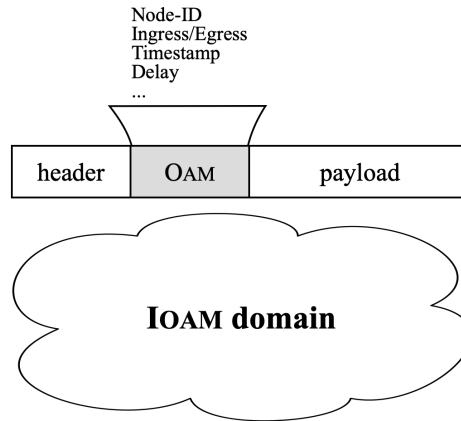


Figure 8.1: Illustration of an IOAM-Domain.

8.1.1 IOAM-Domain

In the context of IOAM, a domain represents a set of nodes that collectively participate in the processing, insertion, modification, and removal of IOAM data within packets, as illustrated in Fig. 8.1. An IOAM-Domain is typically confined to a single administrative or trust boundary, referred to as a “limited domain” [41]. IOAM must be deployed within such limited domains, where participating nodes operate under common administrative control and share consistent configuration, security, and policy enforcement. Within an IOAM-Domain, bounded by its perimeter or edge, packets may be encapsulated with IOAM data that is processed by IOAM nodes along the path. Outside the domain, packets are supposed to be stripped of IOAM information to maintain privacy and prevent leakage of operational data to external networks. The notion of limited domains thus provides a clear operational and security boundary for IOAM deployment. A limited domain that uses IOAM may constitute one or more IOAM-Domains, each distinguishable through the use of IOAM-Namespaces. As noted by RFC 8200, IOAM introduces additional packet overhead, which may impact existing network mechanisms. In particular, operators must account for potential effects on ECMP load balancing. Such effects may arise, albeit unlikely, from the increased packet size introduced by IOAM, which can impact implementations that incorporate packet length into their load-balancing decisions. More significantly, they may result from modifications to packet headers due to encapsulation, which can alter the fields used as input to ECMP hashing. Furthermore, the increased packet size requires careful MTU provisioning to avoid fragmentation or packet loss across all links within the domain.

8.1.2 IOAM-Namespace

IOAM introduces a variety of data fields (*i.e.*, IOAM data), referred to as IOAM-Data-Fields, that can be inserted into packets as they traverse a network. To avoid ambiguity in how this data is interpreted, IOAM defines the concept of IOAM-Namespaces. A namespace serves as a

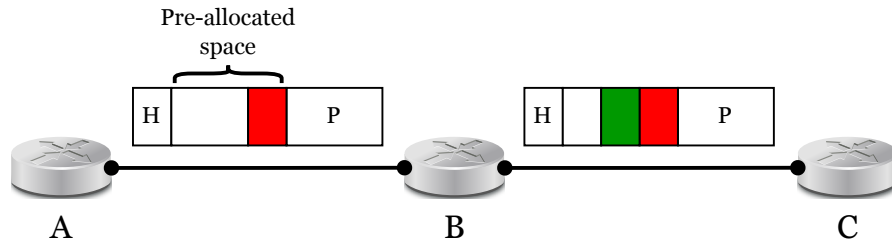


Figure 8.2: IOAM Pre-allocated Trace example. “H” refers to the packet header, while “P” is the payload. Telemetry data (red and green) is inserted in the pre-allocated space. Router A is the *encapsulating* node, router B is a *transit* node, and router C is the *decapsulating* node.

logical scope that provides context for interpreting IOAM data, and is uniquely identified by a Namespace Identifier, which is required in all IOAM options (Sec. 8.2, also referred to as IOAM Option-Types). Each IOAM-Domain operates within a distinct IOAM-Namespace, ensuring that IOAM data from different domains or contexts do not conflict. This design allows multiple IOAM deployments to coexist. Overall, IOAM-Namespace play a foundational role in making IOAM both extensible and operationally practical, by providing the semantic isolation necessary for flexible and interoperable deployment.

8.1.3 Encapsulating, Transit, and Decapsulating nodes

An IOAM-Domain consists of IOAM *encapsulating* nodes, IOAM *transit* nodes, and IOAM *decapsulating* nodes. The role of an IOAM node is performed within a specific IOAM-Namespace. An IOAM node can have different roles in different IOAM-Namespace. Very briefly, an IOAM *encapsulating* node adds one or more IOAM Option-Types into packets for which IOAM is enabled. An IOAM *transit* node reads, writes, and/or processes the IOAM data. An IOAM *decapsulating* node removes IOAM Option-Types from packets. IOAM nodes that add or remove IOAM Option-Types can also update the IOAM data at the same time. In other words, IOAM *encapsulating* or *decapsulating* nodes can also serve as IOAM *transit* nodes at the same time.

8.2 IOAM Option-Types

RFC 9197 defines four Option-Types in IANA’s “IOAM Option-Type” registry [142]: Pre-allocated Trace, Incremental Trace, Proof Of Transit (POT), and Edge-to-Edge (E2E). RFC 9326 defines another Option-Type: Direct Export (DEX) [260]. They are described in this section.

8.2.1 IOAM Pre-allocated Trace

The Pre-allocated Trace is a method for recording per-hop telemetry data within a packet as it traverses an IOAM domain, using a fixed-size space that is pre-allocated to accommodate the data

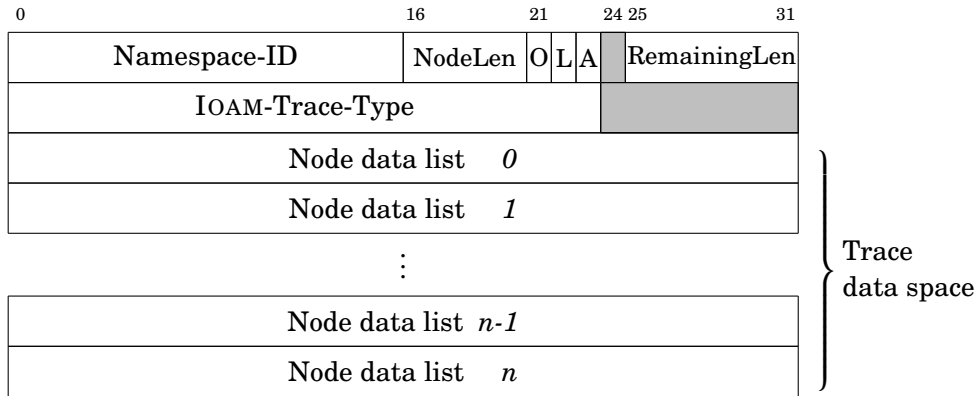


Figure 8.3: IOAM Trace header (both Pre-allocated and Incremental).

records from each hop along the path. As the packet progresses through the network, each IOAM node identifies the next available slot within this pre-allocated space and writes its local data without changing the overall packet length. Fig. 8.2 illustrates how the Pre-allocated Trace works. First, the *encapsulating* node (router A) inserts a Pre-allocated Trace Option-Type, pre-allocates some space and adds its IOAM data (red block in Fig. 8.2). Then, a *transit* node (router B) adds its IOAM data (green block in Fig. 8.2). Finally, the *decapsulating* node (router C) adds its IOAM data and removes the Pre-allocated Trace Option-Type. Fig. 8.3 shows the header format of the Pre-allocated Trace, as specified by RFC 9197, with the definition of the following fields:

- **Namespace-ID** (16-bit) Identifies an IOAM-Namespace.
- **NodeLen** (5-bit) Length of data added by each node in multiples of 4 octets, excluding the length of the “Opaque State Snapshot” (see Table 8.1, bit 22). It depends on the IOAM-Trace-Type field.
- **O-bit** Overflow flag is set when a node detects there are not enough octets left to record its data. It is useful for transit nodes to ignore further processing of the Pre-allocated Trace Option-Type.
- **L-bit** Loopback flag is set to trigger the sending of a packet copy back towards the source (defined by RFC 9322 [210]).
- **A-bit** Active flag is set to indicate that a packet is an active measurement packet rather than a data packet (defined by RFC 9322 [210]).
- **RemainingLen** (7-bit) Specifies the remaining data space, in multiples of 4 octets, for recording the node data before the node data list is considered to have overflowed.

Table 8.1: IANA’s “IOAM Trace-Type” registry [142], which specifies the data types associated to each bit of the IOAM-Trace-Type field in Trace Option-Types (both Pre-allocated and Incremental).

Bit	Description
0	Hop Limit (1 byte) + Node ID (3 bytes)
1	Ingress Interface ID (2 bytes) + Egress Interface ID (2 bytes)
2	Timestamp seconds (4 bytes)
3	Timestamp fraction (4 bytes)
4	Transit delay (4 bytes)
5	IOAM-Namespace specific data (4 bytes)
6	Queue depth (4 bytes)
7	Checksum complement (4 bytes)
8	Hop Limit (1 byte) + Node ID (7 bytes)
9	Ingress Interface ID (4 bytes) + Egress Interface ID (4 bytes)
10	IOAM-Namespace specific data (8 bytes)
11	Buffer occupancy (4 bytes)
12→21	Undefined (4 bytes each)
22	Opaque State Snapshot (TLV, variable-length)
23	Reserved

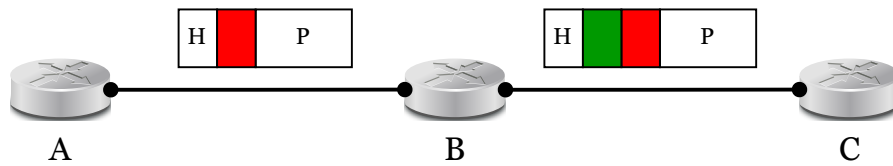


Figure 8.4: IOAM Incremental Trace example. “H” refers to the packet header, while “P” is the payload. Telemetry data (red and green) is inserted in the trace. Router A is the *encapsulating* node, router B is a *transit* node, and router C is the *decapsulating* node.

- **IOAM-Trace-Type (24-bit)** Bit field that specifies which data types are used in the node data list (*i.e.*, which data types are added by each node). Table 8.1 shows the data types corresponding to each bit.
- **Node data list** Variable-length field that contains the list of node data elements determined by the IOAM-Trace-Type field. This is a stack, *i.e.*, the last node to insert data is at the top of the stack, while the first node to insert data is at the bottom of the stack.

8.2.2 IOAM Incremental Trace

The Incremental Trace is conceptually similar to the Pre-allocated Trace, as both share the same header structure (see Fig. 8.3) and collect the same per-hop IOAM data along a path (see Table 8.1).

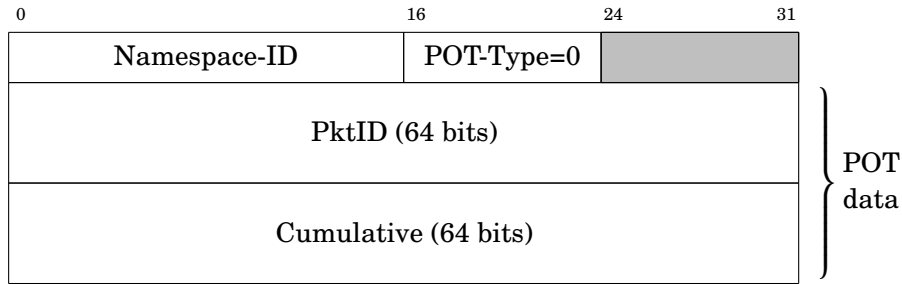


Figure 8.5: IOAM Proof of Transit (POT) Type 0.

The key difference lies in how the trace data is allocated and stored. With the Incremental Trace, each IOAM node (regardless of role) dynamically appends data as packets traverse the path, allowing variable-length tracing without prior knowledge of the path length. This approach is particularly useful when the number of hops is unknown or when minimizing header overhead is important. Its dynamic nature makes it more suitable for hardware implementations. Fig. 8.4 illustrates how the Incremental Trace works. First, the *encapsulating* node (router A) inserts an Incremental Trace Option-Type and adds its IOAM data (red block in Fig. 8.4). Then, a *transit* node (router B) adds its IOAM data (green block in Fig. 8.4). Finally, the *decapsulating* node (router C) adds its IOAM data and removes the Incremental Trace Option-Type.

8.2.3 IOAM Proof Of Transit (POT)

The Proof Of Transit (POT) is used to support path or Service Function Chaining (SFC) [122] verification use cases, *i.e.*, to prove that traffic has traversed a specific path. The mechanism for POT leverages Shamir's Secret Sharing (SSS) [254], which is an efficient secret sharing algorithm for distributing private information (*i.e.*, the secret) among a group. In the context of IOAM, it works by embedding a cryptographic token in packets, which are updated by each IOAM node according to a shared secret. At the egress of the IOAM-Domain, the final token can be validated against the expected result, confirming that the packet has followed a defined path.

RFC 9197 only specified POT-Type 0. Fig. 8.5 shows its header format, with the definition of the following fields:

- **Namespace-ID** (16-bit) Identifies an IOAM-Namespace.
- **POT-Type** (8-bit) Identifies a particular POT variant which specifies the POT data that is included. For this case, it is set to 0 (*i.e.*, the only POT-Type specified by RFC 9197).
- **PktID** (64-bit) Identifies a packet.
- **Cumulative** (64-bit) Cumulative field that is updated at specific nodes by processing per packet PktID field and configured parameters.

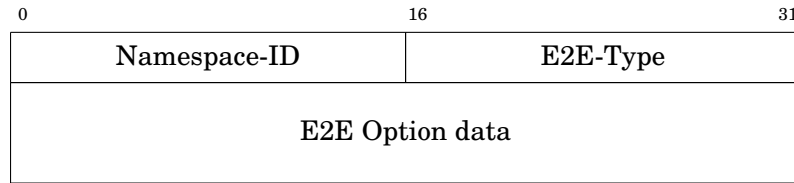


Figure 8.6: IOAM Edge-to-Edge (E2E) header.

Table 8.2: IANA’s “IOAM E2E-Type” registry [142], which specifies the data types associated to each bit of the E2E-Type field in the E2E Option-Type.

Bit	Description
0	Sequence number (64 bits)
1	Sequence number (32 bits)
2	Timestamp seconds (32 bits)
3	Timestamp fraction (32 bits)
4→15	Undefined

Although the POT Option-Type acts as a container for proof of transit mechanisms, these have not reached consensus and are considered dead [32], which makes the POT Option-Type unused and useless at the moment.

8.2.4 IOAM Edge-to-Edge (E2E)

The Edge-to-Edge (E2E) carries IOAM data that is added by the *encapsulating* node and interpreted by the *decapsulating* node. Any *transit* node may process the data without modifying it. Fig. 8.6 shows the header format of the Edge-to-Edge (E2E), as specified by RFC 9197, with the definition of the following fields:

- **Namespace-ID** (16-bit) Identifies an IOAM-Namespaces.
- **E2E-Type** (16-bit) Bit field that specifies which data types are used in the E2E Option data. Table 8.2 shows the data types corresponding to each bit.
- **E2E Option data** Variable-length field that depends on the E2E-Type field.

8.2.5 IOAM Direct Export (DEX)

The Direct Export (DEX), defined in RFC 9326, provides a mechanism for exporting IOAM information from the data plane to an external collector without embedding telemetry data directly within packets. Unlike the IOAM Pre-allocated (or Incremental) Trace, where measurement data accumulates in packets as they traverse the network, DEX serves primarily as a trigger. When an IOAM node processes a packet with the DEX Option-Type, it locally generates IOAM

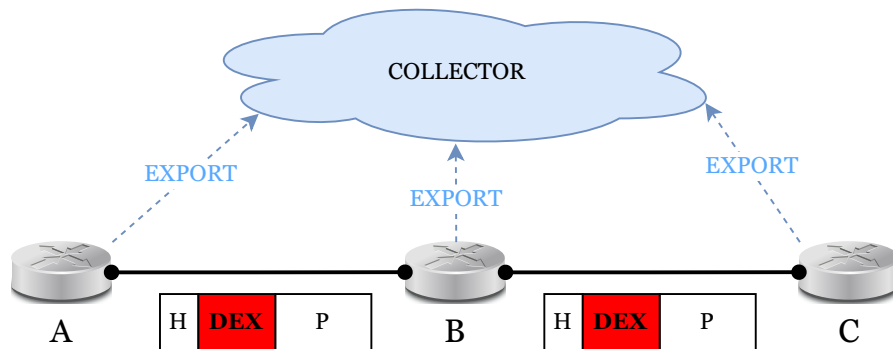


Figure 8.7: IOAM DEX example. “H” refers to the packet header, while “P” is the payload. Router A is the *encapsulating* node, router B is a *transit* node, and router C is the *decapsulating* node.

record containing relevant telemetry, and may export the IOAM data gathered to an out-of-band collector via a separate transport (*e.g.*, IPFIX). Fig. 8.7 illustrates how DEX works. First, the *encapsulating* node (router A) inserts a DEX Option-Type and exports its IOAM data to a collector. Then, a *transit* node (router B) exports its IOAM data to a collector. Finally, the *decapsulating* node (router C) exports its IOAM data to a collector and removes the DEX Option-Type. This approach eliminates the overhead introduced by the IOAM Trace Option-Types, which makes it more scalable and deployable (*e.g.*, reduced probability of an MTU issue). Fig. 8.8 shows the header format of the Direct Export (DEX), as specified by RFC 9326, with the definition of the following fields:

- **Namespace-ID (16-bit)** Identifies an IOAM-Namespace.
- **Extension-Flags (8-bit)** Bit field that specifies the existence of corresponding optional 4-octet fields in the Optional fields. Table 8.3 shows the optional fields corresponding to each bit.
- **IOAM-Trace-Type (24-bit)** Bit field that specifies which data types should be exported. The format of this field is the same as defined for Trace Option-Types (see Table 8.1).
- **Optional fields** List of 4-octet optional fields that depends on the Extension-Flags field.

8.3 IOAM Deployment

IOAM can be deployed [28] and encapsulated in a variety of protocols [4, 18, 27, 29, 30, 102, 207, 294], including Network Service Header (NSH) for Service Function Chaining (SFC), Segment

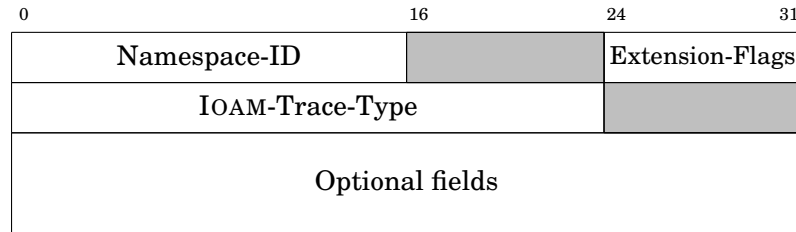


Figure 8.8: IOAM Direct Export (DEX) header.

Table 8.3: IANA’s “IOAM DEX Extension-Flags” registry [142], which specifies the optional 4-octet fields associated to each bit of the Extension-Flags field in the DEX Option-Type.

Bit	Description
0	Flow ID
1	Sequence Number
2	Multicast Branching Node ID (RFC 9630 [261])
3	Multicast Branching Interface Index (RFC 9630 [261])
4→7	Undefined

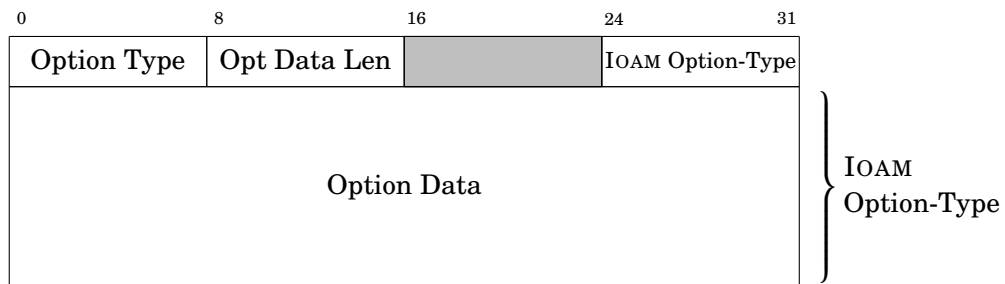


Figure 8.9: IOAM Option: IPv6 Hop-by-Hop and Destination Option format for carrying IOAM data.

Routing and SRv6, Generic Network Virtualization Encapsulation (Geneve), Bit Index Explicit Replication (BIER), IPv6, Generic Routing Encapsulation (GRE), and Generic Protocol Extension for Virtual eXtensible Local Area Network (VXLAN-GPE). However, only the NSH and IPv6 encapsulations have reached consensus and are standardized (*i.e.*, RFC 9452 and RFC 9486 respectively). Given the scope of this document, this section focuses on the IPv6 encapsulation of IOAM.

8.3.1 IPv6 data plane

RFC 9486 specifies how IOAM data is carried within IPv6 packets using IPv6 Extension Headers. It defines the mechanisms for encapsulating IOAM options in the Hop-by-Hop Options Header and Destination Options Header, enabling telemetry data to be collected as packets traverse an IPv6 network. Fig. 8.9 shows the header format of the IOAM option, with the definition of the

Table 8.4: Summary of IOAM options, depending on the IOAM Option-Type carried (*i.e.*, what type of Extension Header is used, and the different values of the “Option Type” and “IOAM Option-Type” fields).

IOAM Option-Type	Extension Header type	IOAM option fields (Fig. 8.9)
Pre-allocated Trace	Hop-by-Hop Options Header	Option Type = 49, IOAM Option-Type = 0
Proof Of Transit (POT)	Hop-by-Hop Options Header	Option Type = 49, IOAM Option-Type = 2
Edge-to-Edge (E2E)	Destination Options Header	Option Type = 17, IOAM Option-Type = 3
Direct Export (DEX)	Hop-by-Hop Options Header	Option Type = 17, IOAM Option-Type = 4

following fields:

- **Option Type** (8-bit) Identifies the type of the option (Chap. 1, Sec. 1.2.1.1). For an IOAM option, it can be 17 (*i.e.*, does not change in transit) or 49 (*i.e.*, may change in transit), as defined in IANA’s “Destination Options and Hop-by-Hop Options” registry [143].
- **Opt Data Len** (8-bit) Length of this option in octets, not including the first two octets.
- **IOAM Option-Type** (8-bit) Identifies the IOAM Option-Type in use, as defined in IANA’s “IOAM Option-Type” registry [142].
- **Option Data** Variable-length field that depends on the IOAM Option-Type field.

IOAM options require alignment on multiples of 4 bytes to ensure that fields specified in RFC 9197 are aligned at a multiple-of-4 offset from the start of the Hop-by-Hop Options Header or Destination Options Header. Also, IPv6 options can have a maximum length of 255 octets. As a consequence, the IOAM option is limited to 255 octets when encapsulated into IPv6.

Table 8.4 summarizes the different IOAM options based on the IOAM Option-Types listed in Sec. 8.2 (*i.e.*, the type of Extension Header used and the value of the “Option Type” and “IOAM Option-Type” fields in Fig. 8.9). The Incremental Trace Option-Type is not listed, and therefore not supported with IPv6, because RFC 8200 does not allow insertion of data directly into the original IPv6 header of in-flight packets. For deployments where the IOAM-Domain is bounded by hosts, IOAM is inserted directly into IPv6 packets. On the other hand, when the IOAM-Domain is bounded by network devices (*e.g.*, routers), IOAM must be encapsulated in an additional outer IPv6 header (and decapsulated later) in order to remain compliant with RFC 8200, which is called an IPv6 tunnel.

8.3.2 Challenges related to IPv6 tunnels

Tunnel-based encapsulation introduces its own set of challenges. In particular, the *encapsulating* node (*i.e.*, the ingress) must determine the correct outer IPv6 destination address, which may

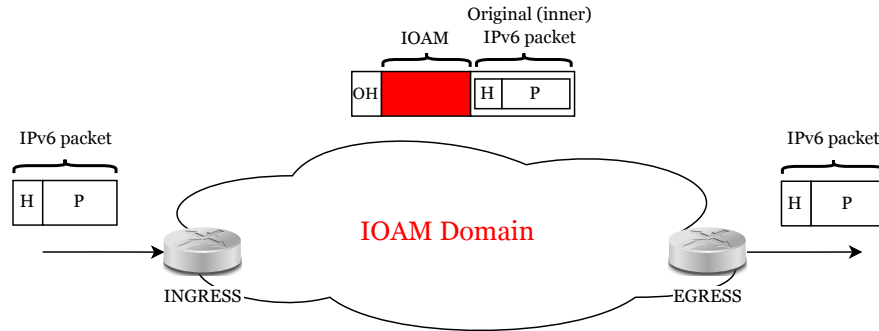


Figure 8.10: Tunnel mode example for IOAM (*i.e.*, IPv6-in-IPv6 tunnel). “H” refers to the packet header, while “P” is the payload. The ingress (*i.e.*, the *encapsulating* node) encapsulates IOAM in an additional outer IPv6 header (*i.e.*, “OH”), and the egress (*i.e.*, *decapsulating* node) removes the outer header and forwards the inner IPv6 packet.

not be trivial in networks with multiple egress points and dynamic routing. Indeed, the ingress must select the egress so that packets with IOAM follow the same path as the original packets. If the ingress selects an incorrect egress, IOAM data may not reach the intended *decapsulating* node or may collect telemetry data for another path. Therefore, the mapping between ingress and egress cannot be assumed to be static, meaning that the outer IPv6 destination address used for encapsulation may vary dynamically, or may need to be selected from a set of possible egresses, adding complexity to IOAM encapsulation. To mitigate some of these challenges, control-plane mechanisms such as the Locator/ID Separation Protocol (LISP) [85] provide a mapping system between endpoint identifiers and routing locators. Such a system allows the ingress to determine the appropriate outer IPv6 destination address for encapsulation in a more deterministic manner. This mapping-based approach can make IPv6-in-IPv6 tunnel encapsulation more reliable for IOAM by ensuring that packets are directed to the correct tunnel egress, even when topology or routing changes occur dynamically. Nevertheless, these mechanisms introduce their own overhead, including additional control-plane complexity, state maintenance, and lookup latency. Moreover, while they assist in resolving egress selection, they do not inherently address other tunnel-related concerns such as MTU management, fragmentation, flow identity preservation, and error reporting to the original source [280]. Fig. 8.10 illustrates how tunnel mode works for IOAM (*i.e.*, an IPv6-in-IPv6 tunnel). The ingress (*i.e.*, *encapsulating* node) encapsulates IOAM in an additional outer IPv6 header (*i.e.*, “OH” in the figure) with a specific egress as the outer IPv6 destination address. The original packet is now the payload of the outer IPv6 header. The egress (*i.e.*, *decapsulating* node) decapsulates the outer IPv6 header and forwards the inner IPv6 packet (*i.e.*, the original IPv6 packet remains untouched).

In contrast, inserting IOAM directly into IPv6 packets offers a simpler solution without the aforementioned issues associated with tunnels, as illustrated in Fig. 8.11. While it is perfectly fine when the IOAM-Domain is bounded by hosts, this approach violates RFC 8200 in other cases

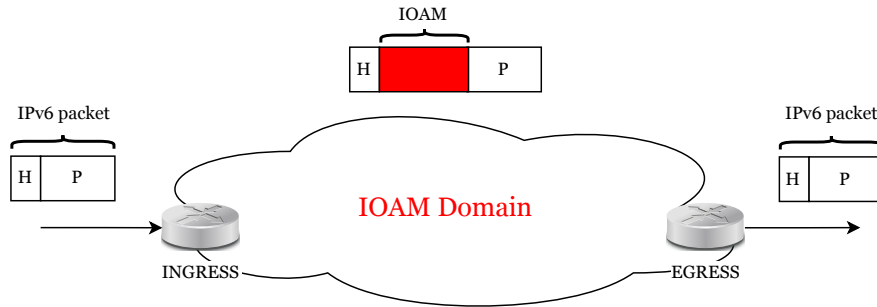


Figure 8.11: Inline mode example for IOAM. “H” refers to the packet header, while “P” is the payload. The ingress (*i.e.*, *encapsulating* node) directly inserts IOAM in IPv6 packets, and the egress (*i.e.*, *decapsulating* node) removes it.

and should be avoided, as already discussed. Some operators may still be tempted to use this mode for convenience, and thus violate RFC 8200 whatever the cost. However, this also has some drawbacks: (i) the *decapsulating* node must remove IOAM from the original IPv6 packets, which is not standardized and is a custom solution (*i.e.*, the *decapsulating* node is not the destination and thus is not allowed to do that); and (ii) in case of problems, ICMPv6 errors are returned to the original source, which would require deep filtering on the edges to prevent IOAM data from leaking outside the IOAM-Domain. An intermediate solution would be to use tunnel mode, but this time with the outer IPv6 destination address identical to the original destination. This avoids most tunnel-related problems, but the *decapsulating* node would still have to remove IOAM from IPv6 packets even though it is not the destination (*i.e.*, not allowed).

In summary, IPv6-in-IPv6 tunnel encapsulation offers a standards-compliant mechanism for carrying IOAM data within user traffic. However, this approach requires careful consideration of tunnel configuration, egress selection, and operational dependencies on control-plane mechanisms such as LISP. The alternative of direct IOAM insertion in IPv6 packets simplifies deployment but conflicts with RFC 8200. Balancing these trade-offs is essential to ensure standards compliance and practical deployability of IOAM in IPv6 networks.

8.4 IOAM Use cases

IOAM can be used in a wide range of operational scenarios, making it increasingly relevant for industry adoption. Typical applications include SLA verification, proof of transit, and geolocation. This section presents two key use cases in detail (Sec. 8.4.1, Sec. 8.4.2) and highlights several additional examples (Sec. 8.4.3).

8.4.1 Smart Traceroute: fault detection and isolation

Traditionally, network fault detection and isolation rely on tools such as ping, traceroute, or Bidirectional Forwarding Detection (BFD). However, in complex environments (*e.g.*, with *microservices*, or networks with a large number of Unequal or Equal-Cost Multipath routes), identifying and isolating faults becomes increasingly challenging. In large-scale networks, fault detection can take tens of seconds, while isolation using traceroute may require several minutes [82, 83]. This is largely due to the probing inefficiency of traceroute, which typically requires at least $2 \times n$ packets for n hops probed (*i.e.*, two packets per hop limit value, one from the source and one from each intermediate hop). Consequently, there is a strong need to enhance network fault detection and isolation through more efficient probing mechanisms, particularly for hypergiants and large Data Center Networks (DCNs) in general.

This context is an excellent example where active network probing based on UDP probes and IOAM can be particularly effective. When connectivity issues arise between individual nodes, IOAM tracing can be selectively enabled to precisely determine where the failure occurs. In this use case, IOAM uses the Pre-allocated Trace Option-Type with the Loopback flag (*i.e.*, L-bit) set, enabling each intermediate hop to return diagnostic information along the reverse path. With this approach, the exact forwarding path can be identified at a substantially lower probing cost, since only $n + 1$ packets are required for n hops probed. Specifically, the source sends a single probe packet with IOAM towards the destination, while each intermediate hop along the path sends a copy of that packet back to the source. Fig. 8.12 illustrates this mechanism with an example. *A* sends a probe packet with IOAM to *F* via the path *A* – *B* – *C* – *F*. As the packet traverses the network, *B* (step 1) and *C* (step 2) each send a copy back to *A*. Because the link between *C* and *F* has failed, *A* does not receive a response from *F*, thus detecting a fault. To further isolate the failure, *A* repeats the same procedure along alternate paths (*e.g.*, steps 3, 4, and 5). When *A* successfully receives a response from *F*, it can infer that the latter is operational and that the failure is confined to the link between *C* and *F* (*i.e.*, based on prior knowledge of topology).

8.4.2 M-Anycast: smart service selection and load-balancing for *microservices*

Consider, for example, a set of highly redundant *microservices* (*e.g.*, video-chunk servers) deployed as containers or Kubernetes (K8s) [270] pods across multiple public clouds. Each instance has its own IPv6 address while sharing the same secondary anycast address. However, access latency, server load, or service availability and liveliness may vary across instances, making it necessary for clients to select the most suitable service endpoint.

With *M-Anycast* [47], this decision is no longer made by the client. Instead, the optimal service at a given time is dynamically determined by the network. More specifically, the *M-Anycast* server acts as a proxy and leverages SRv6 to steer client traffic, while IOAM provides in-band telemetry for optimized service selection. Traditional tools such as ping, traceroute, or BFD cannot achieve this alone, since they lack in-band visibility and path steering capabilities.

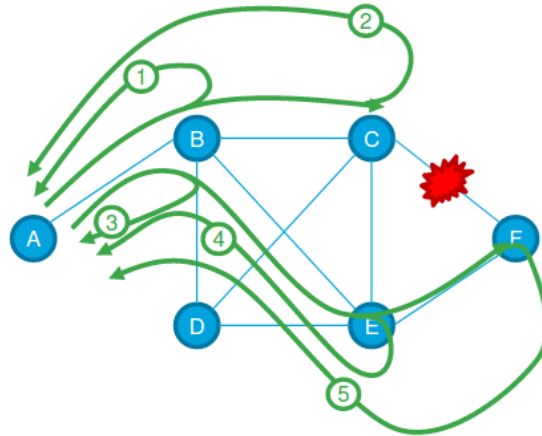


Figure 8.12: Fault detection and isolation with IOAM.

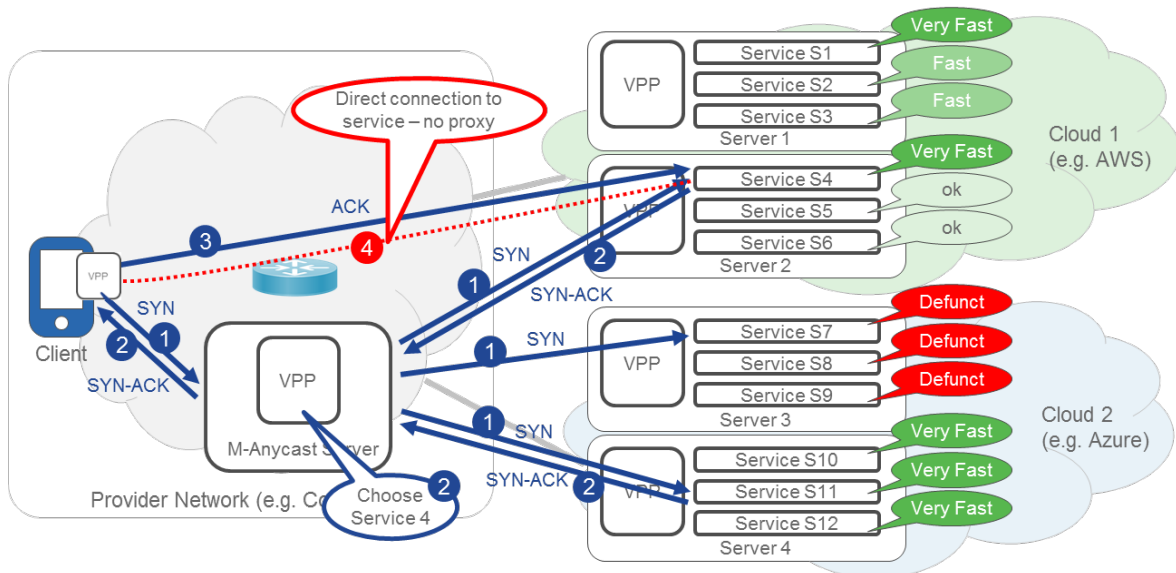


Figure 8.13: M-Anycast: smart service selection and load balancing with IOAM and SRv6. [47]

Fig. 8.13 illustrates how *M-Anycast* works. In step 1, the client wants to establish a TCP session with the service (*i.e.*, the *M-Anycast* server) represented by an anycast address. Thus, the client sends a TCP SYN, which is received by the *M-Anycast* server and then steered towards the chosen set of service instances using Segment Routing Policy [93]. In addition, the *M-Anycast* server adds IOAM to each TCP SYN, so that packet level telemetry can be collected and propagated back to the *M-Anycast* server. Both IOAM and Segment Routing Policy are stripped off by anycast gateways (*i.e.*, services are unaware of Segment Routing and IOAM usage). In step 2, each solicited service replies with a TCP SYN-ACK. Those replies are then steered towards the *M-Anycast* server by anycast gateways, which also add IOAM to each TCP SYN-ACK. The optimal service is selected

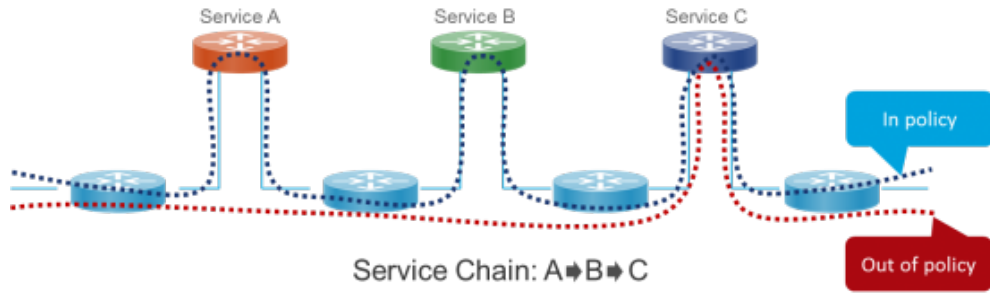


Figure 8.14: Traffic SLAs and service/path verification with IOAM. [46]

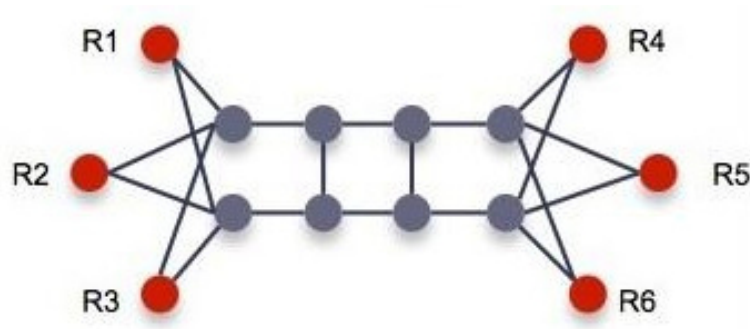


Figure 8.15: Derive network traffic matrix with IOAM. [46]

by the *M-Anycast* server based on the IOAM data received. The *M-Anycast* server then sends the chosen TCP SYN-ACK back to the client, while all other half-open connections are dropped and the connection states are cleaned up. The anycast gateway of the client removes IOAM and the Segment Routing Policy from the TCP SYN-ACK (*i.e.*, clients are unaware of Segment Routing and IOAM usage), and creates a dynamic Segment Routing Policy to steer traffic to the selected service. This means that any follow-up traffic (including the TCP ACK) will be sent directly to the selected service, based on the Segment Routing Policy on the anycast gateway of the client. In step 3, the client confirms the TCP three-way handshake by sending a TCP ACK to the *M-Anycast* server. However, its anycast gateway intercepts the packet and automatically steers it to the selected service (*i.e.*, the *M-Anycast* server is now off-path). The anycast gateway of the selected service receives the TCP ACK, removes Segment Routing, and forwards the packet to the service. In step 4, once the TCP three-way handshake is completed, a TCP session is established between the selected service and the client. Data traffic can now flow directly between them.

8.4.3 Others

Beyond the use cases described above, IOAM also enables several other operational applications. For example, as illustrated in Fig. 8.14, it can be used to verify traffic SLAs and perform service or path verification (*i.e.*, often referred to as Proof of Transit) to ensure that traffic follows a

pre-defined path.

IOAM further supports detailed path tracing capabilities, enabling operators to diagnose complex issues such as ECMP imbalance, latency and delay variations, and to derive both traffic matrices (*e.g.*, Fig. 8.15, which shares similarities with the use case in Sec. 8.4.1) and geolocation information along the forwarding path.

Moreover, by providing fine-grained in-band telemetry on network conditions such as delay and queue depth, IOAM facilitates both congestion detection and adaptive load balancing (*i.e.*, at the queue, interface, and path levels). This enables networks to quickly identify congestion hotspots and automatically react through control-plane or data-plane actions such as rerouting, flow steering, rate adaptation, or queue rebalancing, thereby optimizing flow distribution across large-scale data center fabrics.

Finally, IOAM also serves as a foundation for more advanced observability use cases. One notable example is Cross-Layer Telemetry (CLT), which uses IOAM to correlate both network and application telemetry. This emerging approach is discussed in detail in Chap. 12.

8.5 Comparison with other hybrid measurements

Hybrid measurements were discussed in Sec. 7.3.3. This section compares IOAM to other hybrid solutions, highlighting their similarities, but also their differences.

8.5.1 IOAM vs. Alternate Marking

Alternate Marking was described in Sec. 7.3.3. It shares a similar philosophy with IOAM DEX: both rely on on-path triggers to initiate measurement, while the actual telemetry data is exported off-path. In IPv6 Alternate Marking, special packets carrying the AltMark Option (either in a Hop-by-Hop Options Header or Destination Options Header) act as “color” markers (*i.e.*, triggers), which enable network devices to collect aggregated statistics such as packet counts and delay values. These results are then exported to a collector using a protocol such as IPFIX. Similarly, IOAM DEX uses packets with the DEX Option (either in a Hop-by-Hop Options Header or Destination Options Header) to trigger per-packet telemetry collection at transit nodes. The data is then either aggregated locally or directly exported to a collector (*e.g.*, with IPFIX). The key difference is that Alternate Marking is explicitly designed around interval-based aggregated measurements, while IOAM DEX offers more flexibility, supporting both local aggregation and per-packet export. In addition, Alternate Marking requires coordination between endpoints to ensure consistent “coloring”, which is not needed for IOAM DEX.

8.5.2 IOAM vs. INT

In-band Network Telemetry (INT) was described in Sec. 7.3.3. It is very similar to IOAM in general. INT supports different modes of operation, which correspond to specific IOAM options.

The default mode, INT-MD (also referred to as “classic INT”), is conceptually similar to the IOAM Incremental Trace, as each transit node appends telemetry data directly into packets as they traverse the network. INT-MX, another mode of operation, is conceptually similar to IOAM DEX, as it uses on-path triggers to initiate measurement but exports the collected data to a collector. Overall, the key difference is that IOAM is an IETF standard, whereas INT is specified by the P4 Applications Working Group and is not a standard. In addition, INT requires P4-compatible hardware, which provides flexibility in programmable data planes, and is not directly supported over IPv6. By contrast, IOAM can be deployed on standard IPv6 networks, making it easier to interoperate and deploy in heterogeneous environments.

8.5.3 IOAM vs. HTS

Hybrid Two-Step (HTS) [209] is an IETF draft (*i.e.*, work in progress) and, therefore, not yet a standard. It is an on-path telemetry approach that sits conceptually between IOAM DEX and the IOAM Pre-allocated/Incremental Trace. Like DEX, HTS uses trigger packets to initiate measurement (*e.g.*, by leveraging IOAM itself, or Alternate Marking). Then, HTS uses follow-up packets to collect the actual telemetry data (*i.e.*, similar to the IOAM Pre-allocated/Incremental Trace), reducing per-packet overhead and lowering the impact on the data plane. The key difference is that HTS uses a two-step approach with separate trigger and follow-up packets, whereas IOAM collects telemetry data in a single, one-step process. In addition, HTS can leverage IOAM or Alternate Marking as the underlying mechanism for its operation.

IOAM IMPLEMENTATIONS

Since IOAM remains a relatively recent technology, it continues to attract interest from both the research community and industry. Consequently, multiple implementations have emerged across the networking ecosystem, either developed by open-source projects or integrated by commercial vendors into their network devices. However, these implementations vary widely in terms of supported features and adherence to the IOAM-related RFCs, each being either incomplete, outdated, or non-compliant. This chapter presents a fully RFC-compliant IOAM implementation [154] for IPv6 in the Linux kernel, entirely developed as a key contribution [173] of this research, which serves as the central reference for the chapter and provides the basis for analyzing its features, deployment considerations, and evaluation results. Understanding this work is essential to answering the second Research Question (**RQ-2**), namely “*What is the impact of IOAM on network traffic?*”. In addition, this chapter also discusses other existing IOAM implementations, highlighting their supported features and RFC compliance gaps.

9.1 Linux kernel

Our Linux kernel implementation of IOAM for IPv6 is fully compliant with RFC 9197 and RFC 9486. Specifically, it provides support for the Pre-allocated Trace and is available since kernel version 5.15, with iterative improvements and additional features introduced in subsequent releases. The Incremental Trace is not implemented, as it is not compliant with RFC 8200. Among the other Option-Types, Proof Of Transit (POT) is currently unused due to proof-of-transit mechanisms being considered dead [32], while Edge-to-Edge (E2E) could be implemented in the future but was not the main focus of this research. As for Direct Export (DEX), it is currently under development [104, 105].

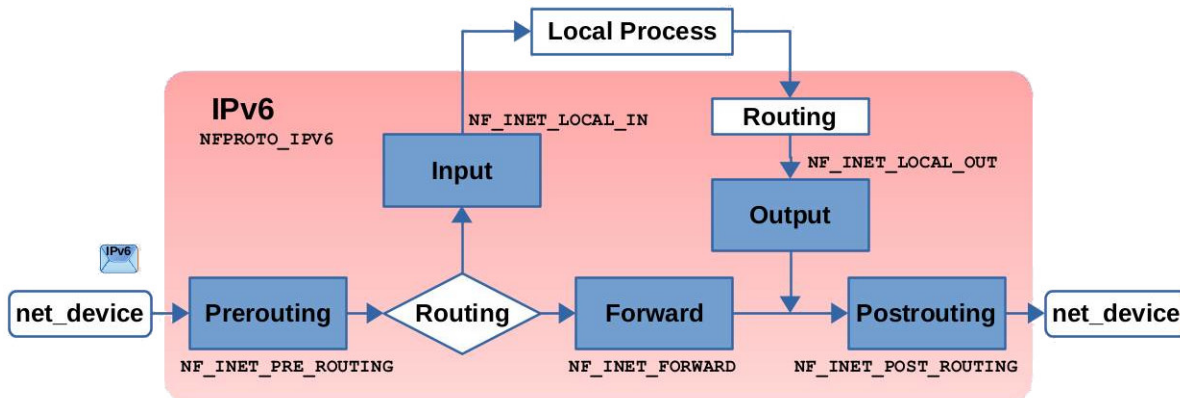


Figure 9.1: The journey of an IPv6 packet through the Linux kernel network stack.

9.1.1 IPv6 packet processing and IOAM architecture

As depicted in Fig. 9.1, an IPv6 packet entering the Linux kernel from a network interface (*i.e.*, *net_device*) is processed at the IPv6 layer, where the IPv6 header is parsed and the Hop-by-Hop Options Header (if present) is handled immediately upon reception, as it may affect subsequent forwarding decisions. The packet then enters the Prerouting hook, where early filtering or mangling can take place before any routing decision is made. The kernel subsequently performs a routing lookup: if the destination is local, the packet passes through the Input hook as it is propagated up the stack toward the corresponding local process; if it is not local, it is forwarded by traversing the Forward hook and then the Postrouting hook before being transmitted out through an interface. For locally generated packets, the path starts at the local process, proceeds through a routing lookup, then the Output hook, and finally the Postrouting hook before transmission. In summary, IPv6 packets take one of three paths: local delivery (input), forwarding, or local output.

Building on this packet traversal, IOAM is integrated at specific points in the receive (RX) and transmit (TX) paths to ensure efficient processing while minimizing redundant operations. On the receive side, IOAM processing (*i.e.*, IOAM header handling and data population) is performed before the Prerouting hook. This placement is motivated by the fact that the IPv6 header and the Hop-by-Hop Options Header are being parsed at this stage, allowing IOAM data to be processed without requiring additional parsing later in the pipeline (*e.g.*, separately in the Input and Forward paths). On the transmit side, IOAM is implemented before the Output hook, aligning with where encapsulation mechanisms such as Segment Routing are applied. This allows IOAM encapsulation (*i.e.*, the insertion of the IOAM header) to be performed at the appropriate stage in the transmit path, after which IOAM data can be populated as needed, while remaining consistent with existing encapsulation mechanisms. In this design, both transit and decapsulating nodes operate on the RX path, while encapsulating nodes operate on the TX path, providing a clear separation between receive-side processing and transmit-side encapsulation.

```

1 struct ioam6_hdr {
2     __u8 opt_type;
3     __u8 opt_len;
4     __u8 :8; /* reserved */
5 #define IOAM6_TYPE_PREALLOC 0
6     __u8 type;
7 } __attribute__((packed));

```

Listing 9.1: IPv6 Option header for IOAM in the Linux kernel.

9.1.2 User space API (uAPI)

The Linux kernel exposes uAPI header files to user space for application use (*e.g.*, to interact with sockets and protocols). In those files, the type `__u8` is used to store an unsigned 8-bit integer in CPU (*i.e.*, host) endianness. There are also `__u16`, `__u32`, and `__u64` for 16-bit, 32-bit, and 64-bit unsigned integers, respectively. For signed integers, there are `__s8`, `__s16`, `__s32`, and `__s64`. The double underscore prefix indicates that these types are intended for uAPI headers, exposed to user space, whereas the equivalent types without the underscores (*e.g.*, `u16` or `s32`) are used internally within kernel code. Additionally, the types `__be16`, `__be32`, and `__be64` are provided to store 16-bit, 32-bit, and 64-bit unsigned integers in big-endian (*i.e.*, network byte order), commonly used for protocol fields and packet headers. There equivalent types to store unsigned integers in little-endian are `__le16`, `__le32`, and `__le64`.

Listing 9.1 shows the uAPI definition of the IPv6 Option header for IOAM, which represents the header specified in Fig. 8.9. Line 2 defines the IPv6 Option Type, line 3 defines the length of the IPv6 Option, and line 6 defines the IOAM Option-Type (*i.e.*, currently set to the value defined in line 5 by default as it is the only Option-Type implemented). Finally, line 7 tells the compiler to not insert any padding between the fields of the structure and to lay them out exactly in the order they are declared.

Listing 9.2 shows the uAPI definition of the IOAM Pre-allocated Trace header, which represents the header specified in Fig. 8.3. For the sake of brevity, only the big-endian representation is illustrated. Line 2 defines the IOAM Namespace-ID, a 16-bit unsigned integer in big-endian (*i.e.*, network byte order). Line 4 defines an 8-bit unsigned integer decomposed into bits: the *NodeLen* field (5 bits), the *Overflow* flag (1 bit), and both the *Loopback* and *Active* flags from RFC 9322 (1 bit each) which are not implemented. Line 9 defines another 8-bit unsigned integer decomposed into bits: an unassigned flag (1 bit), and the *RemainingLen* field (7 bits). Line 12 defines a union to access the IOAM-Trace-Type field in two different ways, for performance and convenience reasons: either as a whole (line 13), or only specific bits (lines 16 → 40). Line 44 defines a constant value to represent the maximum size (in bytes) of the pre-allocated block (*i.e.*, the *data* field in line 45). Finally, line 46 tells the compiler to not insert any padding between the fields of the structure and to lay them out exactly in the order they are declared.

```
1 struct ioam6_trace_hdr {
2     __be16 namespace_id;
3
4     __u8  nodelen:5,
5          overflow:1,
6          :1,          /* unused */
7          :1;          /* unused */
8
9     __u8  :1,          /* unused */
10          remlen:7;
11
12     union {
13         __be32 type_be32;
14
15         struct {
16             __u32 bit0:1,
17                  bit1:1,
18                  bit2:1,
19                  bit3:1,
20                  bit4:1,
21                  bit5:1,
22                  bit6:1,
23                  bit7:1,
24                  bit8:1,
25                  bit9:1,
26                  bit10:1,
27                  bit11:1,
28                  bit12:1, /* unused */
29                  bit13:1, /* unused */
30                  bit14:1, /* unused */
31                  bit15:1, /* unused */
32                  bit16:1, /* unused */
33                  bit17:1, /* unused */
34                  bit18:1, /* unused */
35                  bit19:1, /* unused */
36                  bit20:1, /* unused */
37                  bit21:1, /* unused */
38                  bit22:1,
39                  bit23:1, /* reserved */
40                  :8;      /* reserved */
41         } type;
42     };
43
44 #define IOAM6_TRACE_DATA_SIZE_MAX 244
45     __u8  data[];
46 } __attribute__((packed));
```

Listing 9.2: IOAM Pre-allocated Trace header in the Linux kernel (big-endian representation).

```

1 net.ipv6.ioam6_id
2 net.ipv6.ioam6_id_wide
3 net.ipv6.conf.eth0.ioam6_id
4 net.ipv6.conf.eth0.ioam6_id_wide
5 net.ipv6.conf.eth0.ioam6_enabled

```

Listing 9.3: List of available *sysctls* to configure IOAM on a node.

```
# ip ioam namespace add ID [ data DATA32 ] [ wide DATA64 ]
```

Listing 9.4: Command to configure an IOAM-Namespace on a node.

```
# ip -6 route add PREFIX encap ioam6 [ freq K/N ] [ mode inline | mode encap
tundst IOAM6_TUNDST [ tunsrc IOAM6_TUNSRC ] ] trace prealloc type
IOAM6_TRACE_TYPE ns IOAM6_NAMESPACE size IOAM6_TRACE_SIZE dev eth0
```

Listing 9.5: Command to configure an IPv6 route with IOAM encapsulation.

9.1.3 IOAM configuration

Iproute2 is a modern suite of Linux networking utilities that replaces older tools like *ifconfig*. It provides powerful commands such as *ip* and *tc* to configure interfaces, routing, and traffic control. *Iproute2* primarily communicates with the Linux kernel using Netlink [200, 251], and is used to configure IOAM on a node depending on the IOAM role (*i.e.*, encapsulating, transit, or decapsulating node, as shown in Sec. 9.1.3.1, Sec. 9.1.3.2, and Sec. 9.1.3.3 respectively). There are also *sysctls* to configure different IOAM parameters on a node, as enumerated in Listing 9.3. Line 1 represents the IOAM identifier of the node, while line 2 represents its wide format. Line 3 represents the IOAM identifier of an interface (*e.g.*, *eth0*), while line 4 represents its wide format. Finally, line 5 is a boolean that represents the IOAM status (*i.e.*, 1 if input processing is enabled, 0 otherwise) of a specific interface (*e.g.*, *eth0*).

9.1.3.1 Encapsulating node

The role of the IOAM encapsulating node is to insert IOAM within packets. To configure a node with the IOAM encapsulating role, there are three steps to follow: (*i*) configure the node and interface identifiers (not mandatory, but needed if this data is required in the trace type), as illustrated in lines 1 → 4 of Listing 9.3; (*ii*) configure at least an IOAM-Namespace, as illustrated in Listing 9.4; and (*iii*) configure the IOAM information associated with a route, as illustrated in Listing 9.5. The insertion of the IOAM Pre-allocated Trace relies on Lightweight Tunnels (LWTs) [249], which are explained in detail in Chap. 10.

Listing 9.6 shows an example of configuration for the IOAM encapsulating role. In this case, the IOAM node identifier is set to 1 (line 1). Then, an IOAM-Namespace is created with an identifier of 123 and without namespace-specific data (line 2). Finally, the Pre-allocated Trace is

```
1 # sysctl -w net.ipv6.ioam6_id=1
2 # ip ioam namespace add 123
3 # ip -6 route add 2001:db8:1::/64 encap ioam6 freq 10/100 mode encap tundst 2001:
  db8:f::1 tunsrc 2001:db8:5::1 trace prealloc type 0x800000 ns 123 size 12 dev
  eth0
```

Listing 9.6: Example of commands to configure the IOAM encapsulating role.

```
1 # sysctl -w net.ipv6.conf.eth0.ioam6_enabled=1
2 # sysctl -w net.ipv6.ioam6_id=2
3 # ip ioam namespace add 123 data 0xdeadbeef
```

Listing 9.7: Example of commands to configure the IOAM transit role.

```
1 # modprobe ip6_tunnel
2 # ip link set ip6tnl0 up
```

Listing 9.8: Commands to enable IPv6 tunnel decapsulation for the decapsulating role.

configured to be inserted in 10 packets out of every 100 packets when the destination matches *2001:db8:1::/64* (line 3). The *encap* (*i.e.*, tunnel) mode is used, which adds an outer IPv6 header to encapsulate the original IPv6 packet (in contrast to the inline mode that inserts IOAM directly in the original IPv6 packet). The destination of the IOAM tunnel (*i.e.*, decapsulating node) is set to *2001:db8:f::1*, while the source is set to *2001:db8:5::1*. The latter is optional and, if not specified, the encapsulating node automatically selects one of its IPv6 addresses. The IOAM-Namespaces identifier is set to 123, the configured trace type requires IOAM nodes to add their node identifiers, and the pre-allocated size for the trace is 12 bytes.

9.1.3.2 Transit node

The role of the IOAM transit node is to insert its IOAM data in the pre-allocated trace. To configure a node with the IOAM transit role, there are three steps to follow: (*i*) enable the IOAM input processing on a specific interface, as illustrated in line 5 of Listing 9.3; (*ii*) configure the node and interface identifiers (not mandatory, but needed if this data is required in the trace type), as illustrated in lines 1 → 4 of Listing 9.3; and (*iii*) configure at least an IOAM-Namespaces, as illustrated in Listing 9.4.

Listing 9.7 shows an example of configuration for the IOAM transit role. In this case, the IOAM input processing is enabled on a specific interface (*e.g.*, *eth0*), as illustrated in line 1 of Listing 9.7. Then, the IOAM node identifier is set to 2 (line 2). Finally, an IOAM-Namespaces is created with an identifier of 123 and with the namespace-specific data set to *0xdeadbeef* (line 3).

9.1.3.3 Decapsulating node

The role of the IOAM decapsulating node is to remove IOAM from packets to prevent any

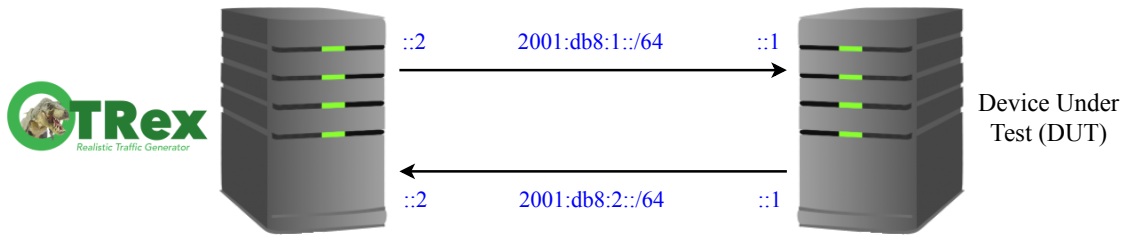


Figure 9.2: Testbed with TReX to evaluate IOAM performance in the Linux kernel.

leakage of sensitive information. When IOAM is used in encap (*i.e.*, tunnel) mode, the decapsulating node strips the outer IPv6 header that carries the IOAM data. In inline mode, this operation is a no-op (*i.e.*, no action is required). The configuration of the IOAM decapsulating role is similar to that of the transit role (see Listing 9.7), with the addition of IPv6 tunnel decapsulation, as illustrated in Listing 9.8. Loading the `ip6_tunnel` module (line 1) and bringing up the default tunnel interface `ip6tnl0` (line 2) enables the Linux kernel’s built-in IPv6-in-IPv6 decapsulation. When this default interface is active, the kernel automatically (*i.e.*, *anonymously*) decapsulates incoming IPv6 tunnel packets without requiring an explicitly configured tunnel endpoint. Note that the command in line 1 is only required if `ip6_tunnel` is compiled as a module, which is the most common case.

9.1.4 Performance evaluation

To evaluate IOAM performance in the Linux kernel, we rely on TReX, which is already described in Chap. 3. Fig. 9.2 illustrates the straightforward testbed: one machine for TReX, and another one for the *Device Under Test* (DUT). Both are equipped with an *Intel XL710 2x40GB QSFP+* Network Interface Card (NIC), each connected port to port 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 isolate a specific function on the DUT and evaluate it, *i.e.*, the IOAM *encapsulating*, *transit*, and *decapsulating* roles separately in this case. The DUT has an *Intel Xeon e5-2630 v3 at 2.40GHz* Central Processing Unit (CPU), with 8 Cores, 16 Threads, and has 16GB RAM. It runs a Linux kernel version 6.9.0. 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, 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).

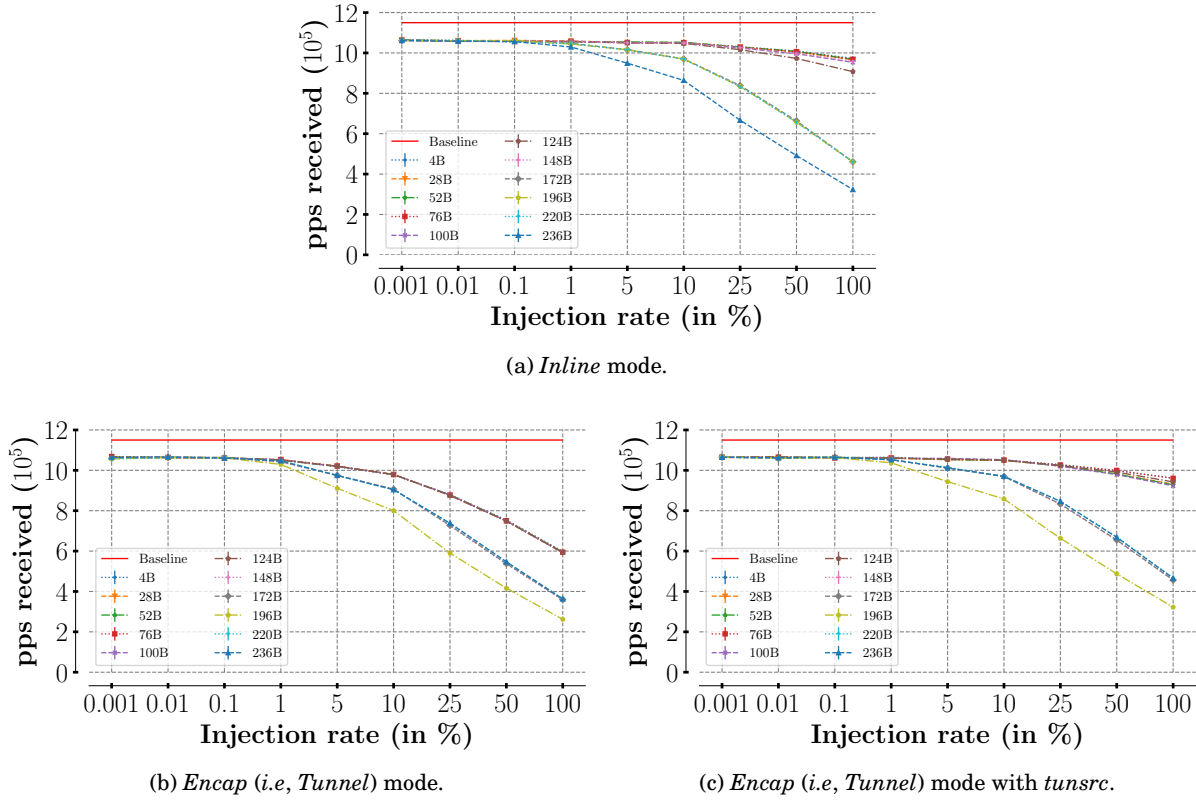


Figure 9.3: Performance of the IOAM encapsulating role, depending on the encapsulation mode, the size of the pre-allocated trace, and the percentage of IOAM traffic.

9.1.4.1 Encapsulating role

Fig. 9.3 shows the performance results of the encapsulating role, for different encapsulation modes. Regardless of the latter, this clearly shows that the encapsulating role has a direct impact on throughput, as all IOAM configurations (*i.e.*, even the smallest trace size at the lowest injection rate) fall below the baseline, which corresponds to a degradation of approximately 10%.

With the inline mode (Fig. 9.3a, *i.e.*, IOAM is added directly to the original IPv6 header), throughput remains stable for trace sizes below 172 bytes, with the first noticeable drops appearing only around 25% of IOAM injection. Beyond that, the degradation becomes more apparent and, at 100% of IOAM injection, throughput sees an additional 5–10% drop. In contrast, for trace sizes of 172 bytes and above, throughput degradation appears much earlier. Drops are already visible at 1% of IOAM injection and then increase rapidly, reaching roughly an additional 50% drop at full IOAM injection. Interestingly, the figure shows that a 236-byte trace performs worse than a 220-byte trace. After investigation, this behavior is due to a bug in the implementation, which is analyzed and resolved in Chap. 10.

The encap (*i.e.*, tunnel) mode (Fig. 9.3b, *i.e.*, the original IPv6 packet is encapsulated in an outer IPv6 header that contains IOAM) performs worse than the inline mode. Drops are already

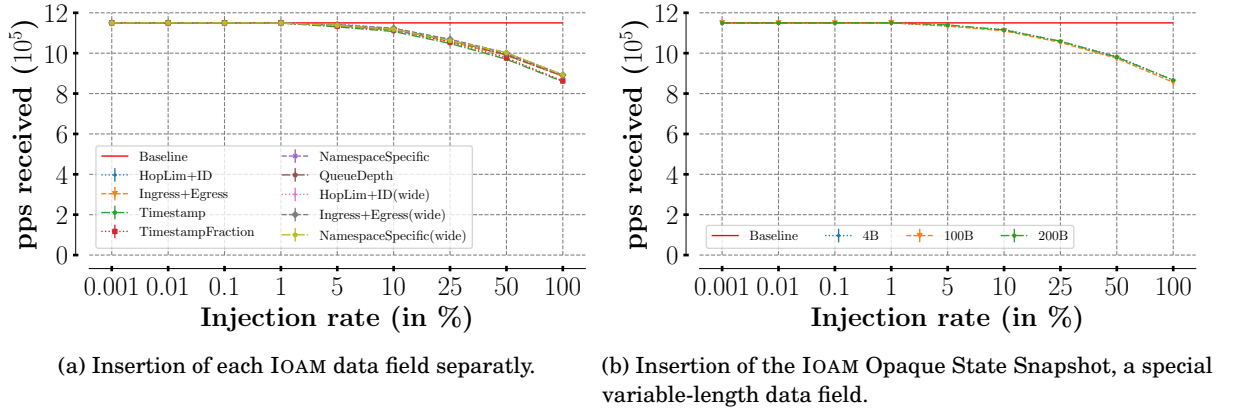


Figure 9.4: Performance of the IOAM transit role, depending on the type of IOAM data field, and the percentage of IOAM traffic.

visible at 1% of IOAM injection for all trace sizes and then increase rapidly, reaching roughly an additional 45% drop at full IOAM injection for trace sizes below 172 bytes, or up to an additional 65% drop for trace sizes of 172 bytes and above. The fact that a 196-byte trace performs worse than a 236-byte trace makes no sense and, after investigation, is also due to the same bug as for the inline mode.

The encaps (*i.e.*, tunnel) mode with *tunsrc* (Fig. 9.3c) performs as good as the inline mode. Thanks to an optimization (*i.e.*, providing the tunnel's IPv6 source address during configuration via *tunsrc*), the data path no longer needs to invoke *ipv6_dev_get_saddr()*, the kernel function responsible for selecting the node's preferred IPv6 source address. This function performs numerous operations (*e.g.*, address-selection logic and policy evaluations) and is therefore relatively expensive.

As advice to operators, the encaps (*i.e.*, tunnel) mode with the *tunsrc* feature should be preferred. The inline mode, although simpler and with similar performance, should be avoided unless operators know what they are doing (*e.g.*, not RFC 8200-compliant for in-transit traffic, risk of data leakage). Also, when forwarding at line rate, operators should pay attention to trace sizes and configure the IOAM injection rate accordingly (*e.g.*, up to 25% maximum for trace sizes below 172 bytes, or up to 1% maximum for trace sizes of 172 bytes and above).

9.1.4.2 Transit role

Fig. 9.4 shows the performance results of the transit role, depending on the IOAM data field to insert and the IOAM injection rate. Fig. 9.4a shows that the IOAM data field with the highest performance cost is the timestamp, as it triggers a CPU clock¹ read for every packet. However, the difference with other IOAM data fields is very small. In general, throughput remains stable up

¹Hardware time-stamping is an optimization but the NIC on the DUT does not support it.

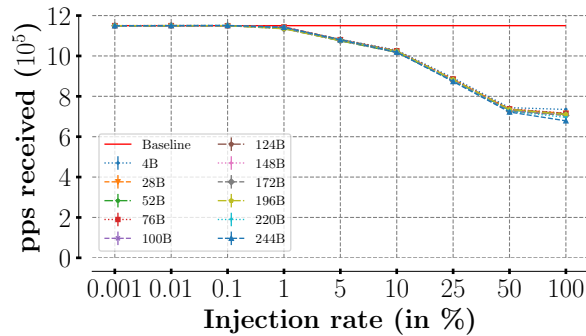


Figure 9.5: Performance of the IOAM decapsulating role, depending on the size of the pre-allocated trace, and the percentage of IOAM traffic.

to 1% of IOAM injection. Beyond that, the degradation becomes more apparent and, at 100% of IOAM injection, throughput drops by approximately 25%. Fig. 9.4b shows that the Opaque State Snapshot (*i.e.*, variable-length) has similar performance to other IOAM data fields, and that the number of bytes to insert has no impact. Overall, the transit role is not the bottleneck compared to the encapsulating role.

9.1.4.3 Decapsulating role

Fig. 9.5 shows the performance results of the decapsulating role, depending on the trace size and the IOAM injection rate. Throughput remains stable until 1% of IOAM injection, where a small drop starts to appear. Beyond that, the degradation becomes more apparent and, at 100% of IOAM injection, throughput drops by approximately 40%. Overall, the number of bytes to decapsulate does not have a significant impact, but the degradation occurs due to the fact that inner IPv6 packets are re-injected in the networking stack after decapsulation. As a consequence, the decapsulating role can be sometimes the bottleneck compared to the encapsulating role (*i.e.*, for trace sizes below 172 bytes with 10% or more of IOAM injection).

9.1.5 Improvements

Several bugs have been fixed (*e.g.*, Chap. 10), and some performance improvements have been implemented (*e.g.*, the *tunsrc* feature). Another work in progress is the support for multi-queue (MQ) network devices in IOAM (*i.e.*, the queue depth data field). The corresponding patch and performance evaluation (see Appendix C) is based on *net-next v6.18-rc6* and, at the time of writing, has not been submitted yet. Overall, the IOAM code path is still actively maintained in the Linux kernel.

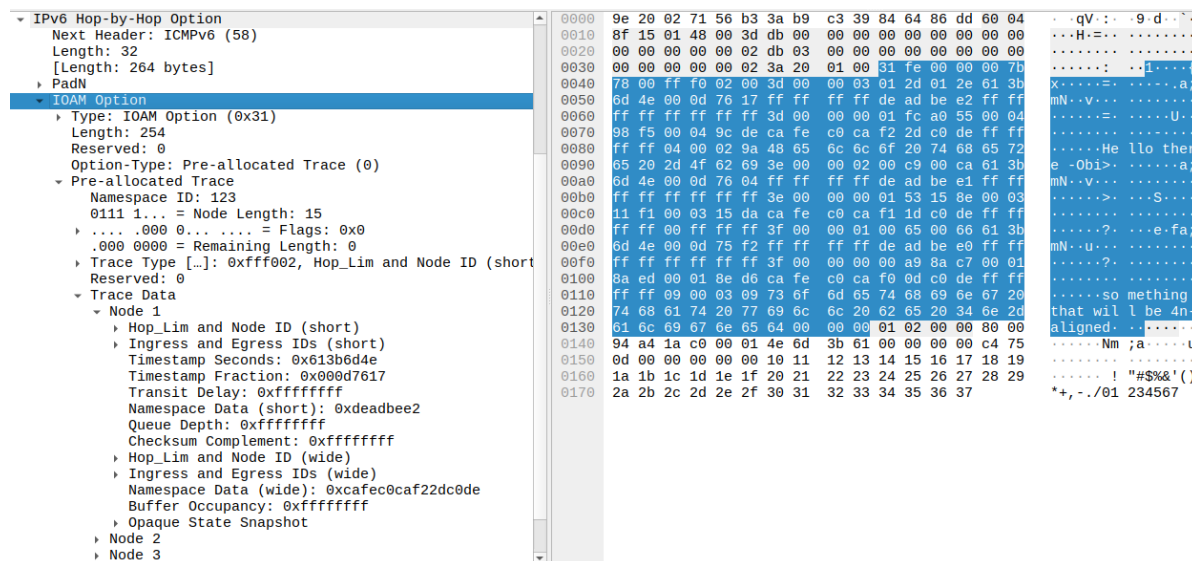


Figure 9.6: Decoding of an IOAM Pre-allocated Trace Option-Type in Wireshark.

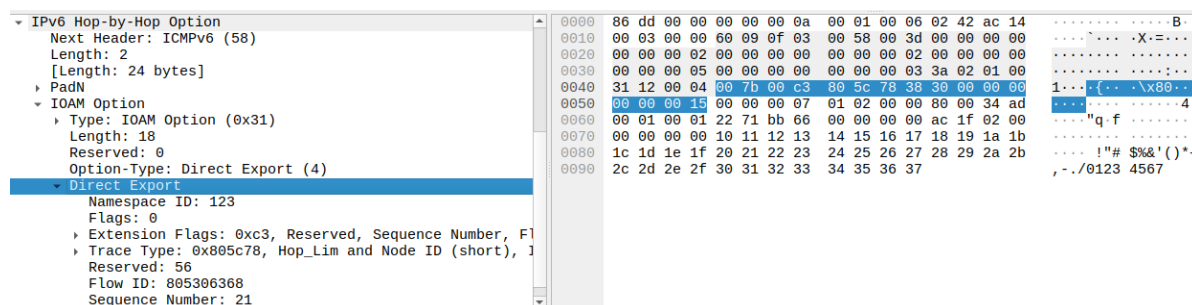


Figure 9.7: Decoding of an IOAM DEX Option-Type in Wireshark.

9.2 Wireshark

Wireshark [298] is a popular tool for capturing and analyzing network traffic. It provides support for the decoding of the Pre-allocated Trace Option-Type (as illustrated in Fig. 9.6) and DEX Option-Type (as illustrated in Fig. 9.7). The IPv6 dissector for the IOAM Pre-allocated Trace was implemented as a contribution [172] to this research and is available since release 4.0, while the IPv6 dissector for DEX was implemented by the University of Liège and is available since release 4.6.

9.3 Vector Packet Processor (VPP)

Vector Packet Processor (VPP) [196] is a user space packet engine that uses efficient batch processing to move packets quickly. It has a modular design and supports many networking

features through plugins. The IOAM plugin implements IOAM over VXLAN-GPE, which is not standardized [29], and IOAM over IPv6. It supports the Pre-allocated Trace, the Proof Of Transit (POT), and the Edge-to-Edge (E2E), and also provides an IPFIX collector to export IOAM data. However, VPP's IOAM implementation is outdated and not compliant with RFC 9197 (*e.g.*, no support for IOAM-Namespaces, limited subset of IOAM-Data-Fields available for the Pre-allocated Trace Option-Type, etc). Therefore, the University of Liège has made efforts to release an up-to-date and compliant version of the implementation [174], but it has not yet been merged upstream.

9.4 Cisco IOS

The data plane implementation in Cisco IOS is only focused on IOAM for IPv6 [50]. It supports the Pre-allocated Trace, the Proof Of Transit (POT), and a sequence numbering feature that is not standardized, and provides a Netflow collector to export IOAM data. However, just like VPP, this IOAM implementation is outdated and not compliant with RFC 9197 (*e.g.*, no support for IOAM-Namespaces, limited subset of IOAM-Data-Fields available for the Pre-allocated Trace Option-Type, etc). IPv6 IOAM is supported on Cisco 1900/2900/3900/3900e Integrated Services Routers.

9.5 Huawei

Huawei provides support for IOAM in CloudEngine 6800/8800/16800 Series Switches [134]. Specifically, they implement the Pre-allocated Trace, Edge-to-Edge (E2E), and Direct Export (DEX). Although the implementation seems to be compliant with RFC 9197 and RFC 9326, IOAM data is added in Layer-4's payload. As a consequence, it is independent of Layer-3 (*e.g.*, IOAM for IPv6, RFC 9486), but it is semantically incorrect considering that transit nodes are involved in IOAM.

9.6 Discussion

Although only a limited number of IOAM implementations are publicly available today, there are persistent indications that several vendors and operators have developed private versions, even if these remain undisclosed. Interest from network operators is clear. For example, Swisscom has already expressed willingness [87] to deploy IOAM (*i.e.*, DEX) as soon as shipped by vendors. From a hardware perspective, IOAM's reliance on TLVs is often considered challenging for high-speed data paths. However, this does not preclude efficient hardware support to overcome software limitations (*e.g.*, the Linux kernel implementation of IOAM), as demonstrated by the successful deployment [89, 189] of SRv6 in modern ASICs. Overall, the future of IOAM looks promising, driven by the emergence of programmable data paths (*e.g.*, INT on Intel Tofino hardware).

BUG DISCOVERY AND FIXING IN THE LINUX KERNEL

In the Linux kernel, IP tunnels are commonly implemented using Lightweight Tunnels (LWTs) [249]. LWTs enable the creation of tunnels on a per-route basis, allowing custom encapsulation mechanisms to be defined through user-specified functions based on the packet destination. Various kernel networking units leverage LWTs, including:

- In Situ Operations, Administration, and Maintenance (IOAM) [31], see Chap. 8.
- Segment Routing for IPv6 (SRv6) [92], a source routing protocol encoding the routing path into IPv6 headers, enabling simplified traffic engineering and flexible network programmability.
- Routing Protocol for Low-Power and Lossy Networks (RPL) [297], a routing protocol designed for resource-constrained networks, particularly in Internet-of-Things (IoT) building optimized multi-hop tree-like structures for reliable communication in dynamic and lossy environments.
- MultiProtocol Label Switching (MPLS) [250], a technology designed to speedup forwarding decisions (through exact label matching instead of longest prefix matching on IP addresses) but is nowadays mainly deployed for Virtual Private Network (VPN) services [219].
- Identifier Locator Addressing (ILA) [129], a way to differentiate between location and identity of a network node.
- Virtual *xfrm* interfaces [185] used for route-based VPN tunnels.

During performance evaluation of our IOAM implementation in the Linux kernel (see Chap. 9), we identified a subtle issue in its behavior, which required further analysis to understand its root cause and to fix it. In fact, the problem lied in the LWTs encapsulation of IOAM. Such a behavior was particularly hard to understand since it occurred only under some conditions that depend on the architecture of the CPU and, to some extent, on the NIC itself. For those specific

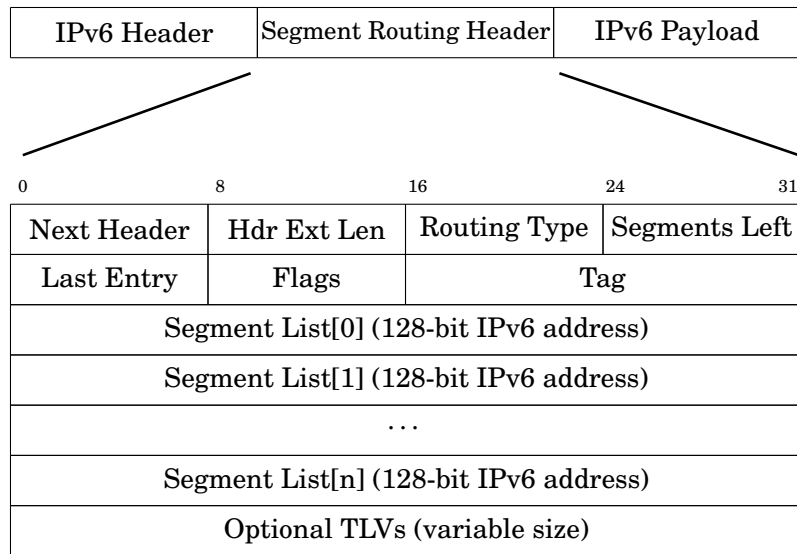


Figure 10.1: Segment Routing Header [91].

cases, and during the LWTs encapsulation in packets, the kernel performs two reallocations of socket buffers instead of a single one, leading to obvious performance degradation. After some investigations, we found similar code patterns causing the exact same issue in the SRv6 and RPL implementations. To quantify the impact on performance, we measured the packet forwarding rate and observed up to 28.8% more drops when this phenomenon occurs.

This chapter explains in details the aforementioned bug and fixes it by proposing a Linux kernel patch to eliminate this inefficiency. The patch replaces the two reallocations with a single, optimized one. In addition, to validate its effectiveness, we conduct a performance evaluation of the three affected protocols, comparing the original implementation with our patched version, demonstrating its ability to improve forwarding efficiency. The patch has been officially merged in the Linux kernel.

10.1 Background

This chapter discusses in detail some very technical aspects of the Linux kernel. In order to make it easier to understand, this section provides some background on SRv6, RPL, the structure of a socket buffer (*i.e.*, *skb*), and the role of the *skb_cow_head()* function in the Linux kernel. As for IOAM, please refer to chapters 8 and 9.

10.1.1 SRv6

Segment Routing [92] is a loose source routing paradigm based on an ordered list of *segments* (*i.e.*, one or more instructions). Each segment can enforce a topological requirement (*e.g.*, pass

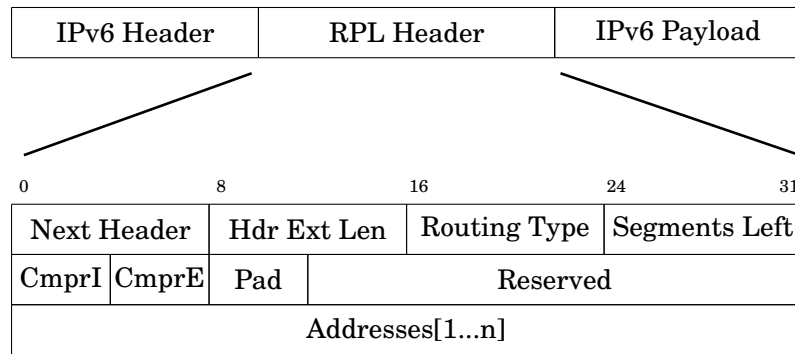


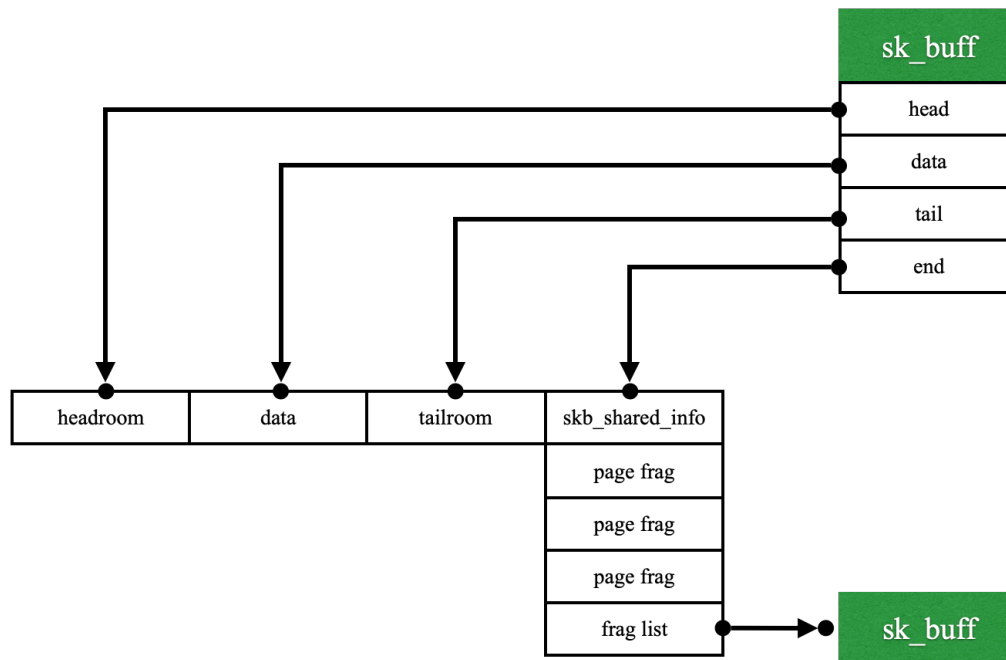
Figure 10.2: RPL Header [136].

through a node or an interface) or a service requirement (*e.g.*, execute an operation on the packet). Over the years, Segment Routing has found a suitable usage for many use cases such as network monitoring, traffic engineering, or failure recovery [287], among others. Two forwarding plane implementations are proposed for Segment Routing: Segment Routing over MPLS (SR-MPLS [15, 86]) and Segment Routing over IPv6 (SRv6 [90]). SR-MPLS requires no change to the MPLS forwarding plane, while SRv6 is based on a Routing Header (*i.e.*, type 4). Each segment is identified by a unique number, a Segment Identifier (SID) implemented as a 128-bit IPv6 address in SRv6, enabling deployments over non-MPLS networks or areas without MPLS, such as data centers. This implementation simplifies deployments as it only requires advertising IPv6 prefixes. SIDs are encoded within a Routing Header (*i.e.*, type 4) known as the Segment Routing Header [91] (see Fig. 10.1).

In the fashion of IOAM, the Linux kernel supports both the *inline* and *encap* (*i.e.*, tunnel) encapsulation modes for SRv6. Additionally, SRv6 features a reduced encapsulation variant (referred to as “Red”) to optimize header size. It also offers a Layer-2 (“L2”) encapsulation variant, where the received frame is encapsulated within the IPv6 packet.

10.1.2 RPL

RPL [297] is a routing protocol for wireless networks, especially suitable for resources-constrained networks such as Internet-of-Things (IoT). It is a Distance Vector Routing Protocol that creates a tree-like routing topology called the Destination Oriented Directed Acyclic Graph (DODAG), rooted towards one or more nodes called the root node or sink node. For downward routing in non-storing mode (one of RPL’s modes of operation), RPL uses a Source Routing header to deliver datagrams, as shown in Fig. 10.2. Just like SRv6, this header contains a list of addresses that the packet must traverse to reach its destination. It comes with a compression mechanism, called Address Abbreviation, designed to reduce the size of the Routing header. If all nodes in the path share a common prefix, only the unique interface identifiers of each node

Figure 10.3: Basic `sk_buff` diagram [180].

are included. Therefore, the RPL header introduces the *CmprI*, *CmprE*, and *Pad* fields to allow compaction of the *Addresses[1...n]* vector when all entries share the same prefix as the IPv6 destination address of the packet. The *CmprI* and *CmprE* fields indicate the number of prefix octets that are shared with the IPv6 destination address of the packet. The shared prefix octets are not carried within the RPL header. The *Pad* field indicates the number of unused octets that are used for padding.

On the contrary to IOAM and SRv6, the Linux kernel only supports the *inline* encapsulation mode for RPL.

10.1.3 Socket Buffer

In the Linux kernel, a socket buffer (*skb*) is a core networking structure used to represent network packets. It acts as a container for the packet's metadata, enabling efficient processing, routing, and management of network traffic. Fig. 10.3 illustrates the layout of the `sk_buff` structure. It consists of several key components:

- `headroom`: Free space available for prepending headers.
- `data`: This area holds the headers and payload.
- `tailroom`: Free space available for adding trailing data if necessary.
- `skb_shared_info`: This structure holds an array of pointers to read-only data.

```
static inline int skb_cow_head(
    struct sk_buff *skb,
    unsigned int headroom
);
```

Listing 10.1: Definition of the *skb_cow_head()* function.

This design allows the kernel to modify headers or add encapsulations while minimizing memory reallocations.

10.1.4 The *skb_cow_head()* function

Listing 10.1 shows the definition of the *skb_cow_head()* function. Its purpose is to make sure that a socket buffer (*i.e.*, the *skb* argument) has at least the desired headroom size (*i.e.*, the *headroom* argument). It is usually used when one only needs to add headers but does not need to modify the data. This function does nothing when there is enough headroom. Otherwise, it reallocates more space to accommodate for the required headroom. In that case, the desired headroom size (in bytes) is aligned to the next multiple of a specific value which is totally dependent on the architecture of the CPU and represents its cache line size. In the Linux kernel, the minimum allowed size for a cache line is 32 bytes and can typically be either 64 (most likely), 128 or even 256 bytes depending on the CPU.

As an example and in order to illustrate, let us assume we want to add 112 bytes to the header of a socket buffer (*skb*), whose current headroom size is 64. Let us also assume a CPU with a 32-byte cache line size. After a call to *skb_cow_head(skb, 112)*, the new headroom size would be 128. The reasoning is the following: (*i*) the CPU cache line size is added to the current headroom size ($64 + 32 = 96$), which is smaller than 112 bytes; (*ii*) the CPU cache line size is re-added to that value ($96 + 32 = 128$), which is now bigger or equal to 112 bytes. Note that after the insertion of the 112 bytes in the header, the new headroom size would be 16 bytes ($128 - 112$).

10.2 Double-Reallocation Issue

This chapter investigates an issue in the LWTs implementation of IOAM, SRv6, and RPL protocols. In normal cases, when an extra header is added to an existing packet, the kernel triggers a reallocation of the socket buffer if its headroom cannot accommodate it. However, in some cases and under specific conditions (see below), the kernel triggers two consecutive reallocations, resulting in performance degradation.

Listing 10.2 shows the problematic code pattern that is common to the LWTs implementation of IOAM, SRv6, and RPL protocols. First, line 1 ensures sufficient headroom in the socket buffer (*skb*) before adding a new header (*hdrlen* bytes), and anticipates the Layer-2 header reconstruction (*mac_len* bytes, *e.g.*, usually 14 for Ethernet). Then, line 2 inserts the new header in the socket

```
1 err = skb_cow_head(skb, hdrlen + skb->mac_len);
2 // add the new header ("hdrlen" bytes)
3 // get "dst" in cache or resolve the new "dst"
4 err = skb_cow_head(skb, LL_RESERVED_SPACE(dst->dev));
```

Listing 10.2: Simplified code pattern that may trigger two reallocations of socket buffers.

buffer, while line 3 gets the destination entry (*dst*) stored in the cache and, if empty, resolves the new destination entry based on the new packet header. This is because the original output device may no longer be the correct one (e.g., an IPv6-IPv6 tunnel encapsulation with different source or destination addresses than the original packet, leading to a different egress interface). Finally, line 4 ensures sufficient headroom in the socket buffer (*skb*) to accommodate the maximum hardware header length of the output device (*dst->dev*), including any extra headroom the NIC may need. The total length returned by the macro `LL_RESERVED_SPACE()` is aligned to 16-byte multiples for machine alignment needs. Indeed, CPUs often take a performance hit when accessing unaligned memory locations. For instance, since an Ethernet header is usually 14 bytes long, network drivers often end up with the IPv6 header at an unaligned offset. The IPv6 header can be aligned by shifting the start of the packet by 2 bytes.

To illustrate a case that would trigger a double-reallocation based on Listing 10.2, let us assume we want to add an extra header (40 bytes) in the socket buffer, whose current headroom size is 22. Let us also assume a CPU with a 32-byte cache line size, and Ethernet (*i.e.*, 14 bytes) as Layer-2. First, in line 1, *hdrlen* is 40 (the extra header size) and *mac_len* is 14 (Ethernet header size). After that, the headroom size becomes 54 (*i.e.*, 22 + 32, the old headroom size plus the cache line size). Then, in line 2, we insert the extra 40-byte header. Therefore, the headroom size is now 14 (*i.e.*, 54 – 40, the old headroom size minus the extra header size). Finally, in line 4, we ensure that the headroom can accommodate the hardware header length (*i.e.*, 16 bytes in this case). Since the current headroom size is too small (14 < 16), a second reallocation is performed. As a result, the headroom size becomes 46 (*i.e.*, 14 + 32, the old headroom size plus the cache line size). The root cause of the double-reallocation comes from the difference between line 1 and line 4 in listing 10.2, where *mac_len* is too generic and often smaller than `LL_RESERVED_SPACE()` due to alignment requirements. A solution to avoid this problem would be to use the latter in both lines, instead of *mac_len*. However, as already mentioned, the output device in line 1 may not be the same after line 3. Therefore, this solution cannot work.

As discussed, the double-reallocation mainly depends on the CPU architecture (*i.e.*, L1 cache line size) and, to a lesser extent, on the NIC (*i.e.*, for alignment requirements and headroom allocation). Headroom allocation depends on the NIC. In our case, the network driver (*i40e*) allocates 192 bytes for the headroom¹. Therefore, before calling `skb_cow_head()` for the LWTs encapsulation, the headroom size of all socket buffers is 206 bytes (*i.e.*, 192 + 14, because the data

¹With *legacy-rx* disabled by default. Otherwise, a cache line sized (*i.e.*, 64-byte for *x86*) headroom is allocated. In both cases, the double-reallocation happens because 192 is a multiple of 64.

Table 10.1: Cases triggering a double-reallocation (x86 architecture, Intel XL710 NIC).

IOAM	
Inline mode	Pre-allocated Trace of 236 or 240 bytes
Encap (<i>i.e.</i> , tunnel) mode	Pre-allocated Trace of 196 or 200 bytes
SRv6	
Inline mode	None
Encap (<i>i.e.</i> , tunnel) mode	For 13, 17, 21, 25, 29, 33, .. segments
Encap (<i>i.e.</i> , tunnel) L2 mode	For 13, 17, 21, 25, 29, 33, .. segments
Encap (<i>i.e.</i> , tunnel) Red mode	For 14, 18, 22, 26, 30, 34, .. segments
Encap (<i>i.e.</i> , tunnel) L2 Red mode	For 14, 18, 22, 26, 30, 34, .. segments
RPL	
Inline mode	None

pointer is moved beyond the 14-byte Ethernet header). Table 10.1 summarizes the cases where a double-reallocation happens for the LWTs encapsulation of IOAM, SRv6, and RPL protocols. It is based on an x86 architecture (*i.e.*, 64-byte cache line size), and an Intel XL710 NIC (*i40e* driver).

Let us take IOAM in Table 10.1 to illustrate the problem. All four IOAM cases actually produce the same overhead (*i.e.*, *hdrlen* provided to *skb_cow_head()*), that is 256 bytes for the entire IPv6 Extension Header (*i.e.*, a Hop-by-Hop Options Header in this case). Indeed, the encap (*i.e.*, tunnel) mode must include the extra IPv6 header (+40 bytes), which is identical to the inline mode in terms of overhead. Moreover, a 236-byte and a 240-byte pre-allocated trace both produce the same overhead, as padding is added to the Hop-by-Hop Options Header. If we apply the code in Listing 10.2 to this case, we have: (i) an extra 256-byte header to add and the 14-byte Ethernet header to rebuild later, (ii) a 206-byte headroom, and (iii) a 64-byte cache line size. After line 1, the headroom becomes 270 (*i.e.*, 206 + 64, the old headroom size plus the cache line size). Then, in line 2, we insert the extra 256-byte header. Therefore, the headroom size is now 14 (*i.e.*, 270 – 256, the old headroom size minus the extra header size). Finally, in line 4, we ensure that the headroom can accommodate the hardware header length (*i.e.*, 16 bytes in this case). Since the current headroom size is too small (14 < 16), a second reallocation is performed. As a result, the headroom size becomes 78 (*i.e.*, 14 + 64, the old headroom size plus the cache line size).

Since the IOAM Pre-allocated Trace is limited to a maximum of 244 bytes, there are no repeated values² that trigger a double-reallocation like for SRv6 and RPL (*i.e.*, maximum 127 segments, meaning 2,040 bytes). Nevertheless, the same logic applies based on the total overhead, *e.g.*, 384 bytes for 21 SRv6 segments in tunnel mode (*i.e.*, extra IPv6 header plus Segment Routing header). Note that the double-reallocation does not happen for SRv6 with the inline mode because it does not fall on the same boundaries (*i.e.*, -40 bytes compared to the tunnel mode with the extra IPv6 header). As for RPL, it comes with a compression mechanism, which makes it theoretically

²There would be more values triggering the issue with *legacy-rx* enabled, depending on the cache line size.

```
1 err = skb_cow_head(skb, hdrlen + dst_dev_overhead(cache_dst, skb));
2 // add the new header ("hdrlen" bytes)
3 // resolve the new "dst" if cache is empty
4 err = skb_cow_head(skb, LL_RESERVED_SPACE(dst->dev));
```

Listing 10.3: Simplified code pattern that mitigates the double-reallocation of socket buffers.

```
static inline unsigned int dst_dev_overhead(
    struct dst_entry *dst, struct sk_buff *skb)
{
    if (likely(dst))
        return LL_RESERVED_SPACE(dst->dev);

    return skb->mac_len;
}
```

Listing 10.4: Code of the new *dst_dev_overhead()* function.

possible to trigger the double-reallocation issue with the inline mode. However, *iproute2* has an input buffer limited to a maximum of 1,024 characters and we were not able to add enough segments to make the total overhead large enough to trigger the issue.

Overall, it is possible to generalize the problem to other CPU architectures: the smaller the cache line size, the more often the double-reallocation of socket buffers would occur (*i.e.*, a 32-byte cache line would trigger the issue twice as frequently compared to a 64-byte cache line). For example, with a 32-byte cache line, the double-reallocation would happen with SRv6 (tunnel mode) for the following number of segments: 11, 13, 15, 17, 19, 21, 23, 25, etc (instead of 13, 17, 21, 25, etc with a 64-byte cache line as in Table 10.1).

10.3 Mitigation Solution

The solution proposed in this chapter leverages the existing cache system in the LWTs implementation of IOAM, SRv6, and RPL protocols, by fetching and using the cache earlier (*i.e.*, before adding the new header). As a reminder, a LWTs encapsulation is attached to a route. Therefore, before a packet matches such a route, the cache, which is supposed to contain the corresponding destination entry (*dst*), and so the corresponding output device, is empty. Once a very first packet matches the route, the LWTs encapsulation is applied to the packet, and the new destination entry is resolved and stored in the cache. After that, the cache is directly used every time another packet matches the same route.

Listing 10.3 shows the simplified code pattern found in listing 10.2, with the mitigation solution applied. Line 1 uses a new function named *dst_dev_overhead()*. The code of this new function is shown in listing 10.4. Its purpose is to return the headroom size required by the output device (for the Layer-2 header and any extra data needed by the NIC) when the cache is

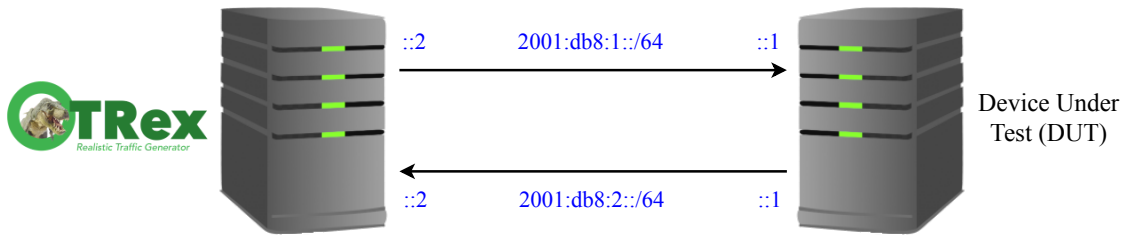


Figure 10.4: Testbed with TReX to evaluate the LWTs implementation of IOAM, SRv6, and RPL in the Linux kernel.

not empty. Otherwise, if the cache is empty, the function returns the generic *mac_len*. Therefore, the cache is now checked before the LWTs encapsulation. The logic behind lines 2, 3 and 4 is the same as previously.

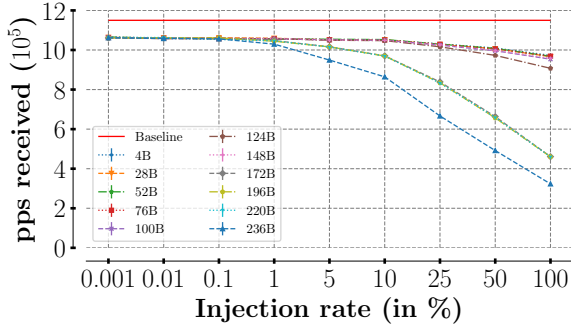
As a result, the first packet that matches a route with an attached LWTs encapsulation would hit an empty cache. Should all specific conditions be met for the double-reallocation to happen, the issue would persist for that first packet only. Indeed, it is impossible to avoid it in that case because the cache is needed, which is not possible for the very first packet. However, subsequent packets are not impacted by the double-reallocation anymore, thanks to the proposed mitigation solution that leverages the cache.

10.4 Performance evaluation

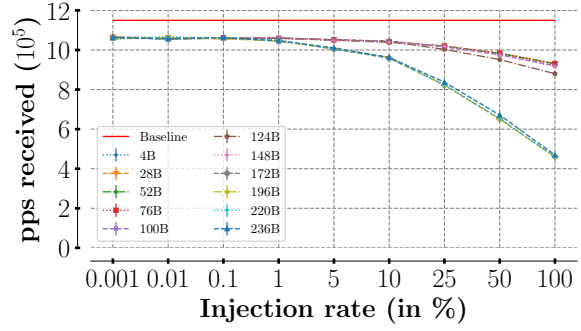
This section describes the methodology to evaluate the LWTs implementation of IOAM, SRv6, and RPL (Sec. 10.4.1), and discusses performance results before and after the patch to mitigate the double-reallocation issue (Sec. 10.4.2).

10.4.1 Methodology

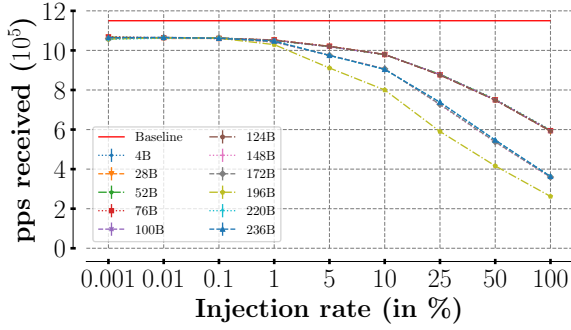
To evaluate performance of IOAM, SRv6, and RPL in the Linux kernel, we rely on TReX, which is already described in Chap. 3. Fig. 10.4 illustrates the straightforward testbed: one machine for TReX, and another one for the *Device Under Test* (DUT). Both are equipped with an *Intel XL710 2x40GB QSFP+* Network Interface Card (NIC), each connected port to port 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 isolate a specific function on the DUT and evaluate it, *i.e.*, the IOAM *encapsulating* role, the SRv6 *encapsulating* role, and the RPL *encapsulating* role separately in this case. The DUT has an *Intel Xeon e5-2630 v3 at 2.40GHz* Central Processing Unit (CPU), with 8 Cores, 16 Threads, and has 16GB RAM. It runs a Linux kernel version 6.9.0. 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



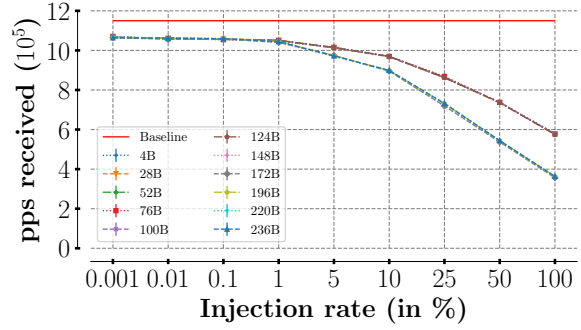
(a) Inline mode – Before mitigation.



(b) Inline mode – After mitigation.



(c) Encap (Tunnel) mode – Before mitigation.



(d) Encap (Tunnel) mode – After mitigation.

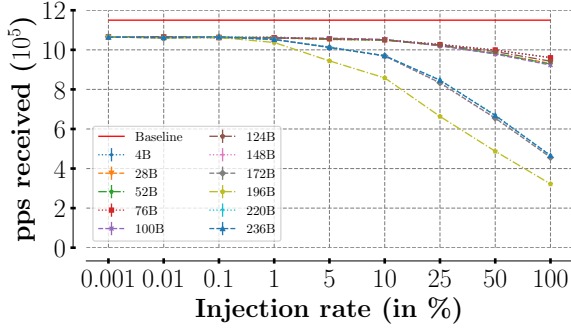
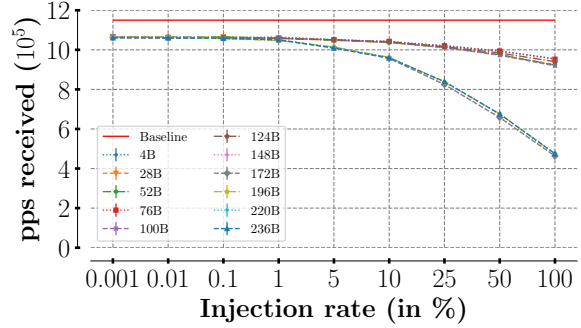
(e) Encap (Tunnel) mode with *tunsrsrc* – Before mitigation.(f) Encap (Tunnel) mode with *tunsrsrc* – After mitigation.

Figure 10.5: IOAM encapsulating role.

for that queue, in order to see the impact on a single core, which is better to compare performance on a common basis. Each experiment (*i.e.*, measurement) lasts 30 seconds and is run 20 times.

10.4.2 Results

Fig. 10.5 shows performance of the IOAM encapsulating role, for each of its encapsulation modes, before and after the mitigation. Fig. 10.5a shows the inline mode before mitigation, where a 236-byte trace performs worse due to the double-reallocation issue. After mitigation (Fig. 10.5b), the

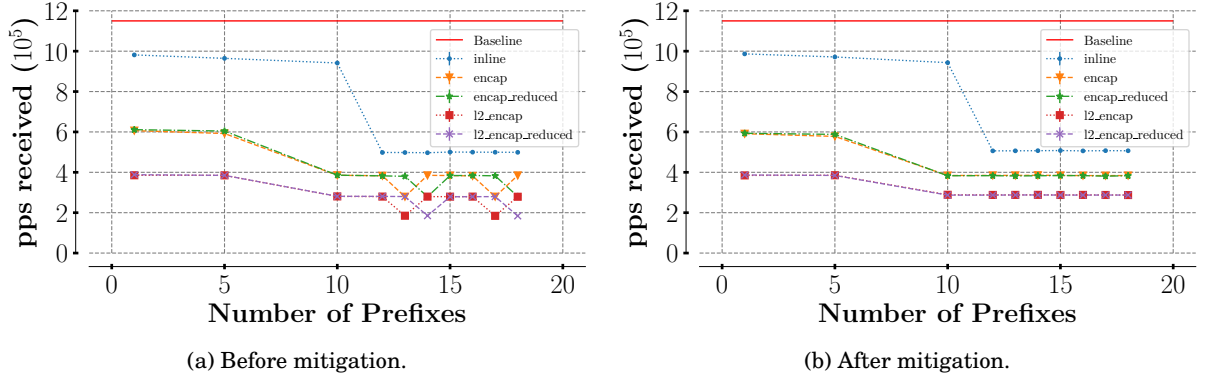


Figure 10.6: SRv6 encapsulating role.

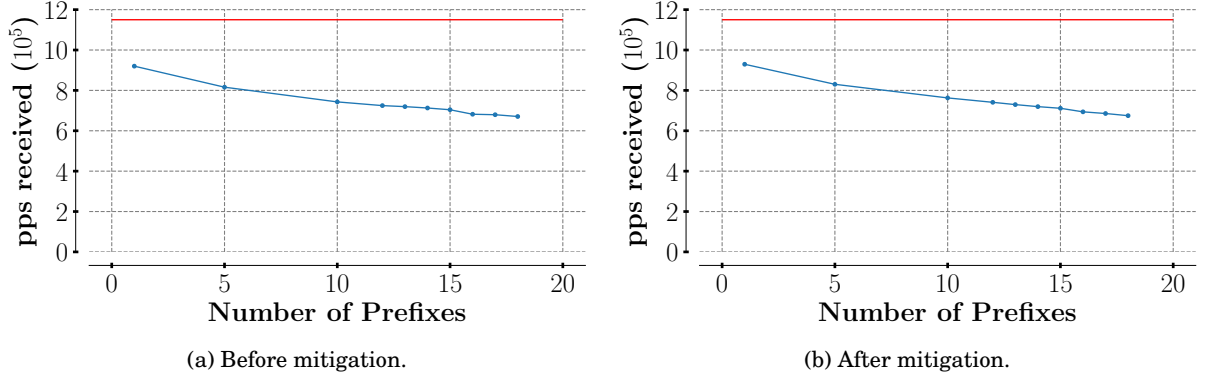


Figure 10.7: RPL encapsulating role.

situation is returning to normal. With the encap (*i.e.*, tunnel) mode before mitigation (Fig. 10.5c, and Fig. 10.5e with *tunsrc*), it is the 196-byte trace that performs worse. After mitigation (Fig. 10.5d and Fig. 10.5f respectively), the situation is also returning to normal.

Fig. 10.6 shows performance of the SRv6 encapsulating role for each of its encapsulation modes, before (Fig. 10.6a) and after mitigation (Fig. 10.6b). While the inline mode is not affected by the double-reallocation issue, a segment count of 13 or 17 for the encap mode in general (vs. 14 or 18 for the reduced encap mode) performs worse. After mitigation, the situation is returning to normal for all its encapsulation modes.

Fig. 10.7 shows performance of the RPL encapsulating role, before (Fig. 10.7a) and after mitigation (Fig. 10.7b), which demonstrates that performance remains consistent with the applied mitigation. This is because RPL is unaffected by the double-reallocation issue, as explained previously.

As a summary, one can conclude that the proposed patch effectively mitigates the double-reallocation issue and does not affect normal cases.

INTEGRITY PROTECTION OF IOAM

Up to now, only few works have been done to secure in-band telemetry protocols such as INT or IOAM. In particular, those works focus on the data plane programmability through INT. For example, Pan et al. [236] use vector homomorphic encryption to design a lightweight telemetry data encryption scheme. Wang et al. [291] rely on INT to identify IP spoofing. They, however, notice that the attacker may tamper with the measurement data, failing the detection. Zhao et al. [305] propose a blockchain-based architecture to incorporate the security triad into the INT architecture. Finally, Kong et al. [187, 188] discuss several attacks INT can suffer and propose Even-Mansour [80] block ciphering and SipHash [13] for integrity validation. With respect to IOAM, Brockners et al. [32] define mechanisms to securely prove that traffic has traversed a defined path. In addition, Brockners et al. [34] perform 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 chapter focuses on integrity protection of IOAM as an attempt to answer the third Research Question (**RQ-3**), namely “*Can we ensure trust in IOAM data?*”. For that, we analyze and leverage what is specified in an IETF draft [33] and provide the following contributions:

- while the IETF draft only specified a single solution, we introduce five potential ways to implement integrity protection of IOAM, and explain how the IETF solution can be improved.
- we discuss the advantages and drawbacks of each solution, leading to a pre-selection of three candidates for implementation.
- we implement [170] those three most promising solutions in the Linux kernel.
- we evaluate those three solutions through extensive performance measurements, leading so to guidance for standardization and a new version of the IETF draft.

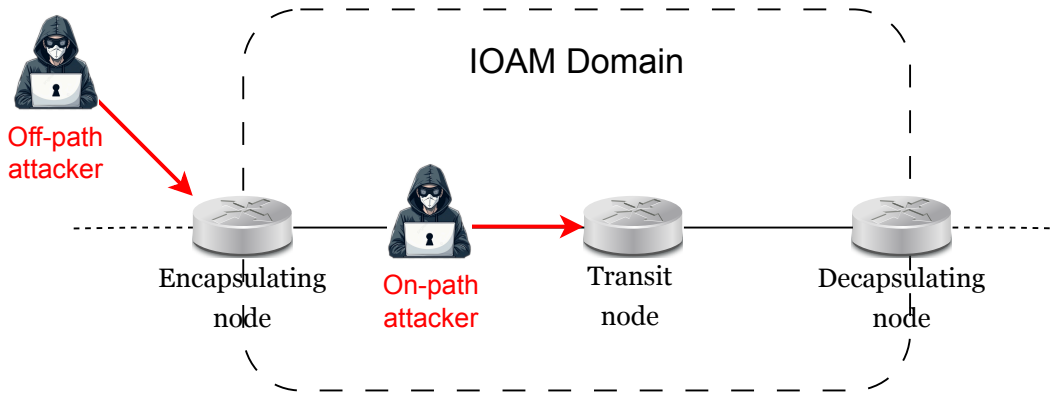


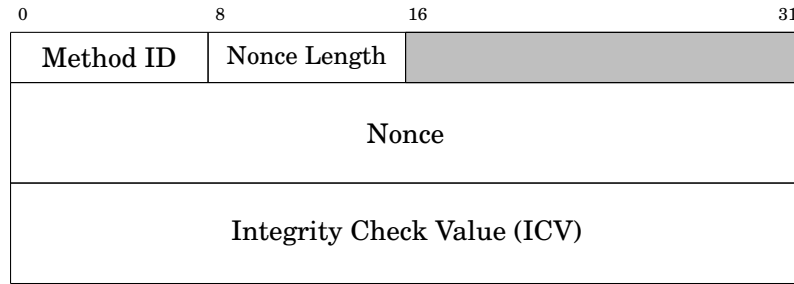
Figure 11.1: IOAM threat model: off-path vs. on-path attackers.

11.1 Threat model

IOAM is designed to operate within a limited domain, *i.e.*, a network under a single administrative control where nodes are assumed to be trusted and where strict boundary filtering prevents IOAM data from being injected into or leaking outside the domain. While this assumption may not always hold in practice due to misconfigurations or partial deployments, this chapter assumes that attacks from the outside are mitigated by proper boundary enforcement. Nevertheless, attacks originating from the inside remain possible, *e.g.*, in the presence of compromised nodes.

Fig. 11.1 illustrates the IOAM threat model, where two adversarial positions are considered: an *off-path* attacker, who can only inject packets, and an *on-path* attacker, who can intercept, modify, drop, or delay packets while they are in transit. For an off-path attacker, the attack surface is primarily limited to the boundaries of the IOAM domain, where forged or replayed packets carrying IOAM data may be introduced. In contrast, an on-path attacker has a significantly broader attack surface, encompassing all nodes along the forwarding path, and can perform a wider range of actions including injection, modification, deletion, replay, and delay of IOAM data. In both cases, the attack surface also extends to the management plane, including configuration mechanisms and out-of-band telemetry export (*e.g.*, collectors). Generally speaking, the impact of successful attacks on the IOAM protocol is an illusion of nonexistent failure or the inability to detect actual ones. Such attacks may also result in a Denial-of-Service by triggering unnecessary processing inside the IOAM domain.

IOAM data is carried in-band and exported out-of-band. It may encode sensitive information such as topology, timing, and performance metrics, which makes it a prime target for attackers. Since this chapter focuses on on-path attacks (*i.e.*, under the assumption that off-path attacks are mitigated through proper boundary enforcement), the primary security objective is the integrity protection of IOAM data. In this context, other security aspects such as confidentiality are considered out of scope, as they would introduce unnecessary complexity for the considered threat model.

Figure 11.2: IOAM Integrity Protection header.¹

11.2 Integrity Protection header

Currently, both IOAM header and IOAM-Data-Fields (*i.e.*, IOAM data) are carried in clear within packets, whatever the Option-Type. As a result, there is no protection against data-altering nodes or middleboxes. Deploying IOAM in an untrusted or semi-trusted environment requires at least integrity protection. An IETF draft [33] extends IOAM in order to carry IOAM header and data with integrity protection. Based on the existing IOAM Option-Types listed in IANA’s “IOAM Option-Type” registry [142], the draft defines new IOAM Option-Types (*i.e.*, new code points in the registry) specifically for integrity protection.

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

The Integrity Protection header is illustrated in Fig. 11.2. The Method ID field defines the algorithm used for computing the integrity. The Nonce Length field specifies the length of the nonce (provided in the Nonce field), in octets. Finally, the Integrity Check Value (abbreviated as ICV in the following) is the integrity value generated by the algorithm specified in the Method ID field.

11.3 Integrity Protection solutions

This section discusses different possibilities for computing and validating the integrity of the IOAM header and data. We compare those solutions (summarized in Table 11.1) and rely on Fig. 11.3 to explain each of them. Ultimately, we want a solution that works the same for all IOAM Option-Types. Therefore, we focus only on the integrity protection of the *Trace* Option-Types (either Pre-allocated or Incremental), due to their uniqueness (*i.e.*, *transit* nodes participate in

¹Both fields “Method ID” and “Integrity Check Value (ICV)” were renamed this way for better semantics.

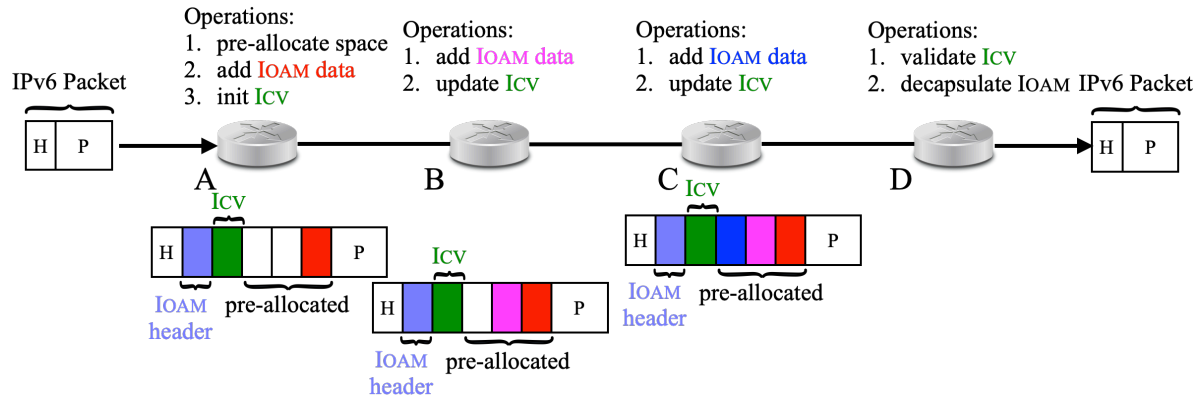


Figure 11.3: Generic example of IOAM data insertion and integrity computation. “H” corresponds to the IPv6 header, while “P” is the IPv6 payload. Node A pre-allocates the space (*i.e.*, Pre-allocated Trace) for IOAM data, adds its own data, and initializes the ICV. Nodes B and C are *transit* nodes adding IOAM data in the pre-allocated space and updating the ICV. Finally, Node D validates the integrity and performs the decapsulation (*i.e.*, removes IOAM).

Table 11.1: Comparison of solutions for integrity protection of IOAM. Solutions 1.x correspond to a single validation at the end (with header check on transit), solution 2 corresponds to a single validation at the end (without header check on transit), solution 3 corresponds to the neighbor (*i.e.*, point to point) validation, and solution 4 corresponds to IPsec. An empty-cell means no support, ~ means partial support, and ✓ means full support. n refers to the number of nodes on a path in the IOAM domain (from the encapsulating node to the decapsulating node; $n=4$ in Fig. 11.3), while p is the node position ($0 \leq p < n$) in the IOAM domain.

Solution	Header		Full Protection	GMAC		
	# ICV	Check on transit		# encap	# transit	# decap
1.a	1	~		1	p	$n-1$
1.b	2	✓		2	2	n
1.c	2	✓		1	2	$n-1$
2	1		✓	1	1	$n-1$
3	1	✓		1	2	1
4	1	✓		1	2	1

integrity protection). Other Option-Types work well with any solution presented in this section, which is not necessarily true for the *Trace* Option-Types.

Fig. 11.3 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. 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 aforementioned IETF draft [33] defines a solution (see Sec. 11.3.1, Solution 1.a) that uses a symmetric key based algorithm for integrity protection, *i.e.*, the Galois Message Authentication Code (GMAC) [70]. 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 [202]), and one could benefit from its hardware implementations. Note that solutions we propose in this section also rely on GMAC, for the same reasons. Equation 11.1 illustrates how GMAC works. It requires three input parameters: a secret (*i.e.*, shared symmetric) key, an Initialization Vector (IV) also called a nonce, and the Additional Authenticated Data (AAD) which represents the data to be integrity protected. It only has one output parameter: the Authentication Tag (ICV), a cryptographic value computed over the AAD using a secret key and a nonce, to check whether the received data has been altered.

$$\begin{aligned} \text{AAD} &= \text{"data to be integrity protected"} \\ \text{ICV} &= \text{GMAC}(\text{key}, \text{nonce}, \text{AAD}) \end{aligned} \quad (11.1)$$

11.3.1 Solution 1: Validation at the End (with header check on transit)

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. The IETF draft [33] defines the following solution.

Solution 1.a: as indicated by Equation 11.2, the *encapsulating* node computes the initial ICV (ICV_0) by using a GMAC operation over the IOAM header and its IOAM data (data_0), with its secret key (key_0) and a generated nonce. Header fields that are mutable must be excluded from the GMAC operation. Each *transit* node i first checks the integrity of the IOAM header by recomputing the integrity chain iteratively. Starting from ICV_0 , it processes the IOAM data in the order they appear in the packet and recomputes each intermediate ICV up to ICV_{i-1} using the same procedure. The recomputed value is then compared against the received ICV to validate the IOAM header integrity. If successful, and as indicated by Equation 11.3, a *transit* node i computes the ICV (ICV_i) by using a GMAC operation over the received ICV (ICV_{i-1} , computed by the previous node) and its IOAM data (data_i), with its secret key (key_i) and the received nonce that was initially generated by the *encapsulating* node. This results in a cumulative chain where each ICV depends on all previously inserted IOAM data. Consequently, this construction strengthens integrity protection. Upon reception, the *decapsulating* node recomputes the entire integrity chain iteratively. Starting from ICV_0 , it processes the IOAM data in the order they appear in the packet and recomputes each intermediate ICV_i using the same procedure. The final recomputed value is then compared against the received ICV to validate integrity.

$$\begin{aligned} \text{AAD}_0 &= (\text{IOAM header} \parallel \text{data}_0) \\ \text{ICV}_0 &= \text{GMAC}(\text{key}_0, \text{nonce}, \text{AAD}_0) \end{aligned} \tag{11.2}$$

$$\begin{aligned} \text{AAD}_i &= (\text{ICV}_{i-1} \parallel \text{data}_i) \\ \text{ICV}_i &= \text{GMAC}(\text{key}_i, \text{nonce}, \text{AAD}_i) \end{aligned} \tag{11.3}$$

This solution has serious drawbacks that could prevent its deployment. Table 11.1 highlights them, as for example: (i) in order to check the header, a *transit* node would have to perform p GMAC operations, with p being the number of previous IOAM nodes which added IOAM data (the *encapsulating* node included); (ii) as a consequence, header check is made optional on *transit* nodes due to its complexity. 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 next solution.

Solution 1.b: as indicated by Equation 11.4, the *encapsulating* node computes the ICV for the header (HICV) over the IOAM header. Header fields that are mutable must be excluded. Then, as indicated by Equation 11.5, the *encapsulating* node computes another ICV for the IOAM data (DICV₀) over its IOAM data (data₀). For each computation, the *encapsulating* node uses a GMAC operation with its secret key (key₀) and a common generated nonce. Each *transit* node i first checks the integrity of the IOAM header by recomputing the HICV (Equation 11.4). If successful, and as indicated by Equation 11.6, a *transit* node i computes the DICV (DICV _{i}) by using a GMAC operation over the received DICV (DICV _{$i-1$} , computed by the previous node) and its IOAM data (data _{i}), with its secret key (key _{i}) and the received nonce that was initially generated by the *encapsulating* node. This results in a cumulative chain where each DICV depends on all previously inserted IOAM data. Consequently, this construction strengthens integrity protection. Upon reception, the *decapsulating* node first checks the integrity of the IOAM header by recomputing the HICV (Equation 11.4) and, if successful, recomputes the entire integrity chain iteratively. Starting from DICV₀, it processes the IOAM data in the order they appear in the packet and recomputes each intermediate DICV _{i} using the same procedure. The final recomputed value is then compared against the received DICV to validate the IOAM data integrity.

$$\text{HICV} = \text{GMAC}(\text{key}_0, \text{nonce}, \text{IOAM header}) \tag{11.4}$$

$$\text{DICV}_0 = \text{GMAC}(\text{key}_0, \text{nonce}, \text{data}_0) \tag{11.5}$$

$$\begin{aligned} \text{AAD}_i &= (\text{DICV}_{i-1} \parallel \text{data}_i) \\ \text{DICV}_i &= \text{GMAC}(\text{key}_i, \text{nonce}, \text{AAD}_i) \end{aligned} \tag{11.6}$$

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 11.1, the *encapsulating* node has to perform two GMAC operations, 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 next solution.

Solution 1.c: as indicated by Equation 11.7, the *encapsulating* node initializes the ICV for *transit* nodes ($TICV_0$), and then computes its ICV (EICV) over the IOAM header and its IOAM data ($data_0$). Header fields that are mutable must be excluded. For the computation of the EICV, the *encapsulating* node uses a GMAC operation with its secret key (key_0) and a generated nonce. Each *transit* node i first checks the integrity of the IOAM header by recomputing the EICV (Equation 11.7). If successful, and as indicated by Equation 11.8, a *transit* node i computes the TICV ($TICV_i$) by using a GMAC operation over the received TICV ($TICV_{i-1}$, computed by the previous node) and its IOAM data ($data_i$), with its secret key (key_i) and the received nonce that was initially generated by the *encapsulating* node. This results in a cumulative chain where each TICV depends on all previously inserted IOAM data. Consequently, this construction strengthens integrity protection. Upon reception, the *decapsulating* node first checks the integrity of the IOAM header by recomputing the EICV (Equation 11.7) and, if successful, recomputes the entire integrity chain iteratively. Starting from $TICV_0$, it processes the IOAM data in the order they appear in the packet and recomputes each intermediate $TICV_i$ using the same procedure. The final recomputed value is then compared against the received TICV to validate the IOAM data integrity.

$$\begin{aligned} TICV_0 &= 0 \\ AAD &= (\text{IOAM header} \parallel data_0) \\ EICV &= \text{GMAC}(key_0, \text{nonce}, AAD) \end{aligned} \tag{11.7}$$

$$\begin{aligned} AAD_i &= (TICV_{i-1} \parallel data_i) \\ TICV_i &= \text{GMAC}(key_i, \text{nonce}, AAD_i) \end{aligned} \tag{11.8}$$

The advantage of Solution 1.c over Solution 1.b is that the *encapsulating* node only performs one GMAC operation, 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 operation. 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 11.1, both Solution 1.b and Solution 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”: while non-IOAM nodes are not trusted, IOAM nodes are forced to

be trusted (*i.e.*, keys must be exchanged between all IOAM nodes to validate the integrity of the header on transit). This can be a problem if an IOAM node (*e.g.*, *transit* node) is compromised. For all those reasons, our first advise would be that no flavor of Solution 1 should be standardized.

11.3.2 Solution 2: Validation at the End (no header check on transit)

This solution is identical to Solution 1.a, except that the *transit* nodes no longer perform integrity checks for the IOAM header.

As shown in Table 11.1, this solution addresses the drawbacks of Solutions 1.x. Here, *transit* nodes only perform one GMAC operation 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: the primary objective is to provide integrity protection for IOAM data, not necessarily the header. In other words, this solution protects the integrity of IOAM data, not the processing of the IOAM Option-Type itself. Of course, some header fields are important and provide context to IOAM data, *e.g.*, *Namespace-ID*, which is common to all IOAM Option-Types. If altered, the IOAM data collected becomes difficult to interpret. This is the reason why the IOAM header is still protected by the *encapsulating* node and checked by the *decapsulating* node, but *transit* nodes do no longer check the IOAM header. Furthermore, this solution is the only one that corresponds to the desired “full protection” scheme, *i.e.*, not a single node has to be trusted (IOAM or non-IOAM ones), except the *decapsulating* node which receives all the keys. Overall, this solution is considered a strong candidate and will be evaluated in Sec. 11.4.

11.3.3 Solution 3: Neighbor Validation

In this solution, integrity validation is performed in a peer-to-peer manner between IOAM nodes. As indicated by Equation 11.9, a sender computes the ICV by using a GMAC operation over the entire IOAM Option-Type (*i.e.*, both the IOAM header and all IOAM data), with its secret key and a generated nonce. Upon reception, a receiver first checks the integrity of the entire IOAM Option-Type by recomputing the ICV with the key of the sender (Equation 11.10). If successful, the receiver then acts as the sender towards the next IOAM node and computes a new ICV (Equation 11.9), with its own secret key and a new nonce. Overall, the *encapsulating* node acts as a *sender*, each *transit* node first acts as a *receiver* and then as a *sender*, and the *decapsulating* node acts as a *receiver*.

$$\begin{aligned} \text{AAD}_{\text{sender}} &= (\text{IOAM header} \parallel \text{IOAM data}) \\ \text{ICV}_{\text{sender}} &= \text{GMAC}(\text{key}_{\text{sender}}, \text{nonce}, \text{AAD}_{\text{sender}}) \end{aligned} \tag{11.9}$$

$$\begin{aligned} \text{AAD}_{\text{receiver}} &= (\text{IOAM header} \parallel \text{IOAM data}) \\ \text{ICV}_{\text{receiver}} &= \text{GMAC}(\text{key}_{\text{sender}}, \text{nonce}, \text{AAD}_{\text{receiver}}) \end{aligned} \tag{11.10}$$

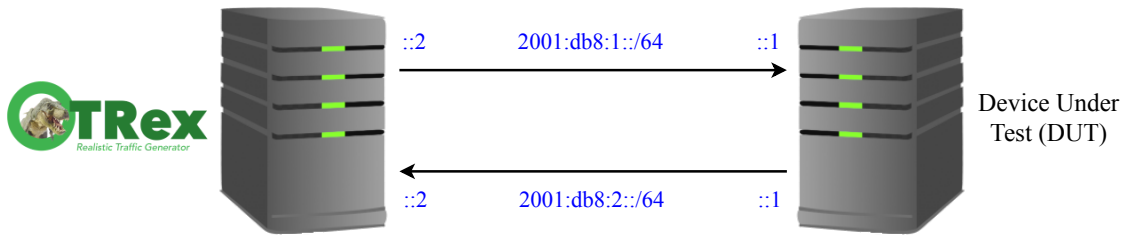


Figure 11.4: Testbed with TReX to evaluate several integrity protection solutions for IOAM.

As shown in Table 11.1, 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 mask mutable fields; (ii) the validation load is shared between IOAM nodes instead of having a single node for validation; (iii) it is elegant, easy and fast to implement. Overall, this solution is considered a strong candidate as a lightweight alternative to Solution 2, and will be evaluated in Sec. 11.4.

11.3.4 Solution 4: IPsec

As shown in Table 11.1, this solution is similar to what Solution 3 proposes, except that it uses existing implementations (*i.e.*, IPsec) without the need to specify a new protocol. The drawbacks are the same as Solution 3, with the additional issues that IPsec tunnels must be explicitly configured between all nodes, not just between IOAM nodes, and that IPsec can cause packets to take a different path, which would kill the benefits of IOAM. Even if this solution seems overkill compared to Solution 3, it is also considered a candidate and will be evaluated in Sec. 11.4.

11.4 Performance evaluation

This section describes the methodology for evaluating previously selected integrity protection solutions (Sec. 11.4.1) and discusses their performance results (Sec. 11.4.2).

11.4.1 Methodology

To evaluate those integrity protection solutions, we rely on TReX, which is already described in Chap. 3.

Fig. 11.4 illustrates the straightforward testbed: one machine for TReX, and another one for the *Device Under Test* (DUT). Both are equipped with an *Intel XL710 2x40GB QSFP+* Network Interface Card (NIC), each connected port to port 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 isolate a specific

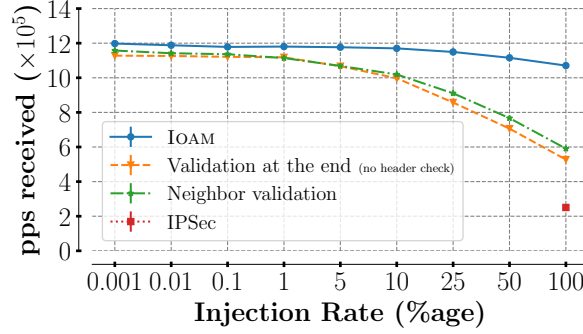
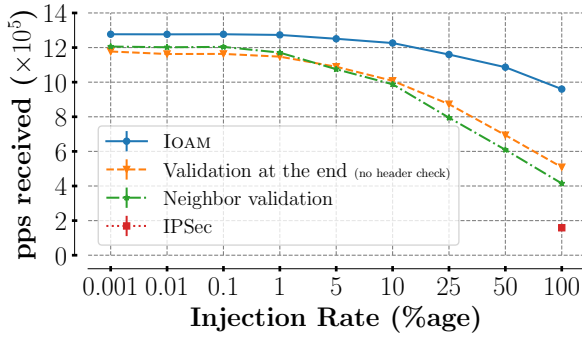
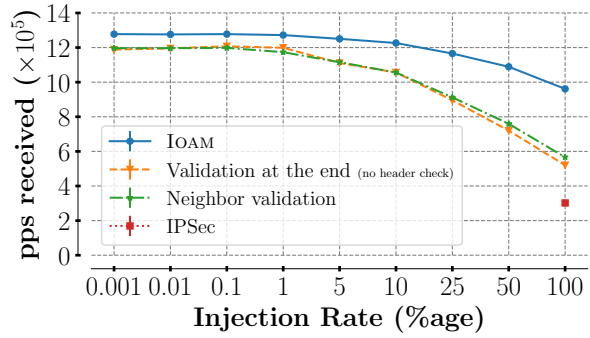
(a) *Encapsulating node (A in Fig. 11.3).*(b) *Transit node (B or C in Fig. 11.3).*(c) *Decapsulating node (D in Fig. 11.3).*

Figure 11.5: Performance of IOAM integrity protection solutions. The line labeled IOAM corresponds to vanilla IOAM, without any integrity protection, and serves as performance baseline for each role.

function on the DUT and evaluate it, *i.e.*, the IOAM *encapsulating*, *transit*, and *decapsulating* roles separately in this case. The DUT has an *Intel Xeon e5-2683 v4 at 2.10GHz* Central Processing Unit (CPU), with 16 Cores, 32 Threads, and has 64GB RAM. It runs a Linux kernel version 6.9.0. 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 the integrity protection 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). Integrity protection solutions 2, 3, and 4 (Sec. 11.3) were implemented by leveraging the existing IOAM implementation [154, 162] in the Linux kernel.

11.4.2 Results

The machine used for the DUT differs from the one used for previous measurements and is more powerful. Consequently, all figures below, specifically the blue “IOAM” curves, are shifted upward compared to the measurements in Chap. 9, reflecting a vertical offset that represents the improved performance. The overall shape of the curves remains proportional, indicating that the performance trend is consistent regardless of the machine used.

Fig. 11.5 shows performance results per IOAM role, respectively for an *encapsulating* node (Fig. 11.5a), a *transit* node (Fig. 11.5b), and a *decapsulating* node (Fig. 11.5c), and so for each integrity protection solution we evaluated, *i.e.*, Solution 2 (“*Validation at the end – no header check on transit*”, Sec. 11.3.2), Solution 3 (“*Neighbor validation*”, Sec. 11.3.3), and Solution 4 (“*IPsec*”, Sec. 11.3.4). 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. 11.5a shows that both Solution 2 and Solution 3 share similar performance, which is somehow expected since they perform one GMAC operation for this role (*i.e.*, equal to the offset from the baseline). However, Solution 3 seems to perform slightly better, especially when IOAM is injected in more than 10% of the traffic. Such a difference can be explained by the simplicity of Solution 3 and the way it is implemented, compared to Solution 2 which is a little 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 operation. Regarding Solution 4, 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 scenario.

Then, let us focus on the *transit* role. Fig. 11.5b shows that both Solution 2 and Solution 3 have, again, similar performance, up to 10% of IOAM injection. After that, one can clearly see that Solution 3 becomes less efficient, which is explained by the fact that Solution 2 performs only one GMAC operation while Solution 3 performs two GMAC operations. Both solutions are suitable for line-rate when IOAM is added up to 1% of the traffic. Without any surprise, Solution 4 is terrible for 100% and can be explained by the fact that the following steps (in that order) are performed for each packet: (i) one GMAC operation (integrity validation), (ii) one IPv6 decapsulation due to the IPsec tunnel mode, (iii) one IPv6 re-encapsulation, and (iv) one GMAC operation (to compute the ICV).

Finally, let us focus on the *decapsulating* role. Fig. 11.5c shows that both Solution 2 and Solution 3 share, again, similar performance. This may seem strange since Solution 2 performs a variable number of validations (*i.e.*, GMAC operations) depending on the number of nodes in the trace, and should be slower. In fact, as a common baseline in this figure, only one validation (*i.e.*, GMAC operation) is performed by the *decapsulating* node with Solution 2, which also corresponds

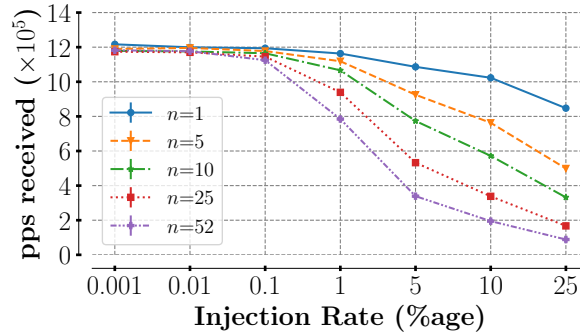


Figure 11.6: Solution 2: Validation at the end (no header check on transit) – zoom on the *decapsulating* node for a variable number of validations (*i.e.*, GMAC operations).

to its best case scenario (*i.e.*, similar to Solution 3). Therefore, Fig. 11.6 shows the impact of a variable number of validations (*i.e.*, GMAC operations) for the *decapsulating* role with Solution 2. One interesting observation is that the maximum number of validations (*i.e.*, 52, due to space constraint with IOAM) 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 usually corresponds to the average IOAM domain length.

11.5 Discussion

Based on the above, Solution 2 (“*Validation at the end – no header check on transit*”, Sec. 11.3.2) is definitely the best compromise and the IETF draft [33] should standardize it instead. While the solution proposed in that draft is similar in some aspects, Solution 2 comes with key differences to make it deployable, and is also the only one that provides “full protection”. With this solution, integrity validation is one-step in some cases (*e.g.*, with POT Type-0 or E2E Option-Types), where only the *encapsulating* node updates the ICV, not *transit* nodes. For other cases where *transit* nodes also update the ICV (*e.g.*, with Trace Option-Types), integrity validation is variable and depends on the number of nodes in the trace. Regarding Solution 3 (“*Neighbor validation*”, Sec. 11.3.3), it could be standardized later as a lightweight, probably simpler, and intermediate solution, if operators have a need. However, it should be kept in mind that it does not provide “full protection” since keys are exchanged between all IOAM nodes, making any compromised IOAM node capable of impersonating other IOAM nodes. As for Solution 4 (“*IPsec*”, Sec. 11.3.4), it should only be used for inter IOAM domain use cases, to securely transfer IOAM data between IOAM domains. It is neither effective nor suitable for intra IOAM domain use cases, as IPsec tunnels must be explicitly configured between all nodes, not just between IOAM nodes, and as IPsec can cause packets to take a different path, which would kill the benefits of IOAM. Moreover, Solution 4 does not provide “full protection” since IOAM transit nodes still need to be trusted.

Another important recommendation for the IETF draft is that it should not make mandatory

for a *decapsulating* node to perform integrity validation. Instead, it should make it optional and propose an alternative where integrity validation is delegated off-path, in order to avoid slowing down the data plane. This would allow operators to build optimized infrastructures, *e.g.*, with an off-path load balancer to distribute integrity validation tasks across multiple dedicated nodes.

Finally, the IETF draft should add more text to discuss critical aspects such as interoperability and security. For example, GMAC requires not to reuse a nonce with the same key. The IETF draft should clarify how it can be achieved securely.

Appendix D contains the latest version [34] of the IETF draft (version 19, at the time of writing), which includes all the recommendations and improvements suggested in this chapter. This IETF work has just been approved and is about to become an IETF RFC .

CROSS-LAYER TELEMETRY (CLT) FOR ADVANCED OBSERVABILITY

Modern cloud-native applications tend to rely on *microservices*, especially hypergiants such as Netflix due to their complex architectures [19, 25]. *Microservices* are a way of designing applications as a collection of small, independent services that perform a specific function, and they usually communicate with each other over the network using lightweight protocols such as gRPC/google Remote Procedure Calls (gRPC) [114, 121]. Each *microservice* can be developed, deployed, and scaled independently, which makes the overall system more flexible and easier to maintain than a single, large monolithic application. A single request in an application can invoke a lot of *microservices* interacting with each other. As a matter of fact, it is difficult to monitor and isolate a problem, *e.g.*, a slowdown of a service. This is why Application Performance Management/Monitoring (APM) is useful for *microservices* troubleshooting, as discussed in Chap. 7. Thanks to APM, it is possible to observe and understand the whole chain of events in a complex interaction between *microservices*. However, it becomes less useful when the problem is not related to applications but rather located in the network itself. Therefore, having both network and application telemetry is a must, but there is also a need to have them combined, which comes with big challenges.

To answer the fourth Research Question (**RQ-4**), namely “*Can we achieve unified observability that correlates network and application layers?*”, this chapter introduces Cross-Layer Telemetry (CLT) [150] as a means to match network telemetry (*i.e.*, provided by IOAM) with application telemetry (*i.e.*, provided by APM traces), offering a full network stack visibility to APM tools. With CLT, APM traces are correlated with corresponding traffic that carries network telemetry, providing a better view and a faster root cause analysis to APM tools in case of a network issue (*i.e.*, APM tools are now able to *see* the network). There is not much related work in this area, due to the complexity of integration without touching libraries, socket APIs, or network stack layers,

making standardization complicated. For example, Vogt et al. [288] propose SATS (*i.e.*, Server and Application Telemetry System), a lightweight framework to standardize and invert the in-band network telemetry paradigm by enabling servers and applications to push fine-grained state data to in-network applications. Their approach moves application state into the network, whereas CLT integrates and propagates network telemetry upward and across layers (*i.e.*, the opposite). Also, Fan Yi [301] introduces *Athena*, a measurement framework that correlates telemetry from the 5G physical layer up to the application layer, which is similar to CLT but very specific to 5G.

The remainder of this chapter is organized as follows: Sec. 12.1 describes in detail the implementation of CLT, its architecture, and associated challenges; Sec. 12.2 evaluates each component of CLT separately; Sec. 12.3 evaluates the overall performance of CLT based on a use case; finally, Sec. 12.4 discusses main achievements and next steps.

12.1 CLT Implementation

With APM tools, operators can receive useful information on applications thanks to application telemetry, which generally refers to Layer-7 in the network stack. However, when including APM traces and session-level context, application telemetry extends into Layer-6 and Layer-5 as well. Let us assume an HTTPS request to be monitored inside an application. Now, let us picture a situation in which an operator notices an abnormal execution time of that request. With APM, it is difficult to exactly understand why it happens. Worse, if the problem is not application related but, rather, on the link or on intermediate hops (*e.g.*, due to congestion), the operator would be stuck wondering why the request takes so long as the application side looks fine. In a perfect world, a complete solution would show how the request progresses hop-by-hop through the network and identify (potential) bottlenecks. Indeed, by correlating application telemetry (*i.e.*, APM traces) with network telemetry (*i.e.*, corresponding packets with IOAM), operators would have a far more complete tool to deal with problems. This is exactly the purpose of CLT, as it makes the entire network stack (*i.e.*, from Layer-2 to Layer-7) visible to APM tools, instead of the classic application visibility.

12.1.1 Version History

As discussed in Chap. 7 (Sec. 7.1.1), an APM trace is represented by a unique trace identifier and contains one or more spans, each with its own span identifier. To achieve full-stack visibility in APM tools, network telemetry must be correlated with APM traces by injecting the trace and span identifiers into the data plane. This, however, is not trivial. For example, identifiers can change within a single TCP connection because multiple requests (and thus multiple traces or spans) may be multiplexed over the same socket. Conversely, a socket may be reused across clients, allowing distinct traces to traverse the same socket. Therefore, injecting these identifiers only at socket creation cannot guarantee correct per-request correlation. Instead, injection must



Figure 12.1: Call graph of *send* system calls for IPv6 and Layer-4 in the Linux kernel.

occur at send time, on a per-request basis. One possible approach is to extend the *syscall* interface with new *send* primitives that accept trace and span identifiers. The drawback is that this would require coordinated changes across the kernel, the C standard library (*i.e.*, *libc*), programming language runtimes, and applications, due to the fact that it would violate network stack layers by trying to provide Layer-7 knowledge to Layer-4. A less disruptive (and likely more practical) alternative is to convey the identifiers to the kernel with Netlink before sending the data. This trade-off is discussed in Sec. 12.4.

The first version of CLT, let us call it the socket annotation technique, uses a Netlink call to pass the identifiers to the kernel, which in turn copies them into the corresponding socket. Each packet going out of this socket would therefore carry the identifiers. Although this solution was working really well, it had a non-negligible downside, namely packets that could be marked with the wrong identifiers due to, *e.g.*, congestion in a queue. Indeed, a packet for the annotation *X* could still be in the queue when another annotation *Y* erases the previous one, which would make the packet annotated with *Y* instead of *X* as soon as it is sent out.

The second version of CLT (the current one), let us call it the packet annotation technique, deals with the aforementioned problem by annotating a packet as soon as it is created for a specific socket. This version is described in the next sections of this chapter.

12.1.2 Kernel Space

For CLT, the kernel must be modified. The patch is available [150] and is quite straightforward. Briefly, the *sock* structure (*i.e.*, representation of a socket) and the *sk_buff* structure (*i.e.*, representation of a packet, called a socket buffer) are extended with two fields: (i) a 128-bit field for the trace identifier; and (ii) a 64-bit field for the span identifier. A Netlink call is added to pass the trace and span identifiers from user space to the kernel, right before sending data, and so for a specific socket file descriptor. With the first version of CLT, these identifiers were only copied into the *sock* structure, leading to the problem described above. Now, with the second version, as soon as a packet is created to send data through the annotated socket, these identifiers are also copied into the *sk_buff* structure (*i.e.*, the packet itself is annotated). Fig. 12.1 shows the call graph of all *send* system call primitives. With IPv6, both *tcp_sendmsg* for TCP and *udp6_sendmsg* for UDP are functions that create socket buffers. Thanks to that, we make sure that a socket buffer (*i.e.*,

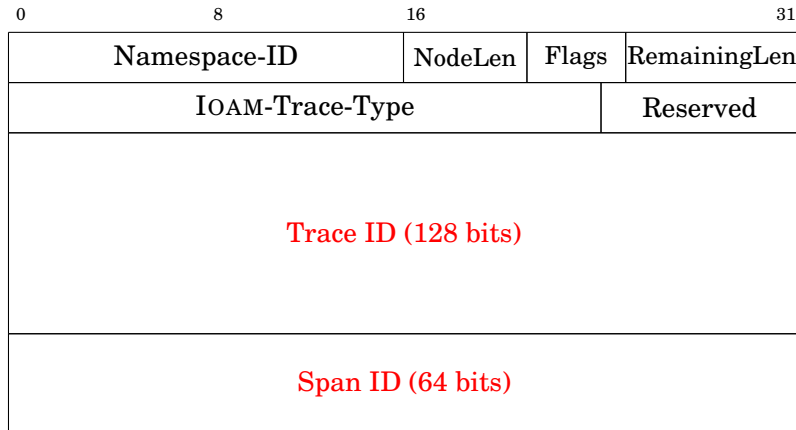


Figure 12.2: Enhanced IOAM Pre-allocated Trace header.

```

1 span = tracer.start_span('test') # OpenTelemetry tracing
2 CLT.enable(sockfd, span.trace_id, span.span_id)
3 resp = https.request('GET', '/test') # Uses the underlying socket "sockfd"
4 CLT.disable(sockfd)
5 span.end()

```

Listing 12.1: Simplified example of a Python application that uses OpenTelemetry for tracing, with additional annotations (lines 2 and 4) for CLT.

packet) is annotated directly with the corresponding trace and span identifiers, instead of waiting for the packet to be sent out through the socket.

CLT relies on IOAM for network telemetry, in particular on the IOAM Pre-allocated Trace, to carry data (*e.g.*, timestamp, queue size, buffer occupancy, node or interface identifiers, etc) within packets for each hop on the path. As such, it is the perfect candidate to also carry both APM identifiers (*i.e.*, trace and span), which kills two birds with one stone. Fig. 12.2 illustrates how the IOAM Pre-allocated Trace header is extended to support both trace and span identifiers. This trade-off is discussed in Sec. 12.4.

12.1.3 User Space

A CLT client library is provided [150] for user space applications. It is currently available for Python and Golang but could be easily implemented in other programming languages as needed. The CLT client library encapsulates the new Netlink call logic to pass the trace and span identifiers to the kernel. Listing 12.1 shows an example of code to trace an HTTPS request with OpenTelemetry in a Python application. To use CLT, only two additional lines of code are required, *i.e.*, lines 2 and 4. The only downside is that most programming languages do not easily expose the socket layer to the application layer (*e.g.*, when using an HTTPS object to send requests) due to how the network stack is abstracted. Therefore, in application code, the underlying socket

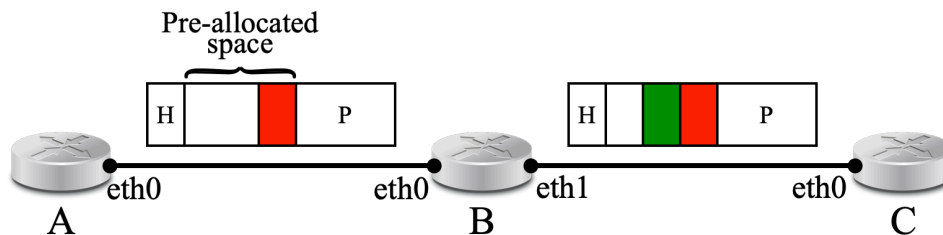


Figure 12.3: IOAM Pre-allocated Trace example. “H” refers to the packet header, while “P” is the payload. Telemetry data (red and green) is inserted in the pre-allocated space. Router A is the INGRESS of the IOAM domain, while C is the EGRESS.

```

1 # sysctl -w net.ipv6.conf.eth0.ioam6_enabled=1
2 # ip ioam namespace add 123
3 # ip -6 route add 2001:db8:1::1/128 encap ioam6 trace prealloc type 0x800000 ns
   123 size 12 dev eth0

```

Listing 12.2: IOAM command-line configuration.

must either be manually injected into such objects or retrieved from them to access the socket file descriptor. This kind of workaround tends to be easier in some programming languages than in others.

As previously mentioned, CLT relies on the IOAM Pre-allocated Trace to carry both APM identifiers (*i.e.*, trace and span) along with network telemetry. Therefore, it must be configured with the *iproute2* tool (*i.e.*, the “ip” command) that has support for the IOAM Pre-allocated Trace. As a reminder, Fig. 12.3 illustrates how the IOAM Pre-allocated Trace works. To set it up, the operator must configure an IOAM domain between the three nodes. In this case, IOAM is only used from A (INGRESS) to C (EGRESS) but not on the reverse path, meaning IOAM must be allowed for both *B.eth0* and *C.eth0*. This is easily achieved with *sysctl* configuration on B and C (line 1 in Listing 12.2). Then, an IOAM namespace (*e.g.*, 123) is created on A, B, and C (line 2 in Listing 12.2). Finally, A must be configured to insert an IOAM Pre-allocated Trace in IPv6 packets when C (*e.g.*, *2001:db8:1::1*) is the destination (line 3 in Listing 12.2). Once configured, if A receives an IPv6 packet with destination C, it adds the IOAM Pre-allocated Trace with the IOAM namespace 123, and inserts its IOAM data corresponding to the red block in Fig. 12.3. When B receives an IPv6 packet with an IOAM Pre-allocated Trace, it inserts its IOAM data if the IOAM namespace is 123 (the green block in Fig. 12.3). When C receives an IPv6 packet with an IOAM Pre-allocated Trace, it inserts its IOAM data if the IOAM namespace is 123. Since C is the destination, the entire IOAM Pre-allocated Trace is thus available.

12.1.4 Telemetry Data Collection and Processing

In this telemetry ecosystem, we also provide an IOAM agent to collect and report any IOAM Pre-allocated Trace received on a specific interface. In short, this is a per-interface IPv6 packet sniffer that filters a Hop-by-Hop Options Header with an IOAM Pre-allocated Trace. After parsing, each IOAM Pre-allocated Trace is transformed into an IOAM Trace API object [152], which is specified using Protocol Buffers v3 [115]. The IOAM agent can be run in two different modes: *output* or *report* (default mode). The output mode prints IOAM Trace API objects in the command-line interface, while the report mode sends them to a collector with gRPC. The original versions of the IOAM agent are implemented in Python [151] and C++ [293], using a simple *AF_PACKET* socket. An improved version of the IOAM agent is implemented in Go [292] and leverages *PF_RING* [228], which is a high-performance packet capture framework. Their performance is compared in Sec. 12.2.

Finally, we provide an IOAM collector [153], which is a Golang interface between the IOAM agent and a specific backend collector (in this case, Jaeger). Its role is to augment an existing span by attaching IOAM data sent by the IOAM agent. The augmented span is then forwarded to the backend collector, which correlates it with the original span. Depending on specific requirements, the IOAM collector may be implemented in other programming languages or integrated with backends other than Jaeger.

12.1.5 Architecture

Fig. 12.4 illustrates the architecture of CLT, and shows the interactions between each component. Based on the code snippet in Listing 12.1, let us illustrate what happens step by step. In line 1, the application uses the OpenTelemetry client library (the “OTEL client” green box in Fig. 12.4) to create a new span called “test” and to add it to the current APM trace. Then, in line 2, the application uses the CLT client library (the “CLT” blue box in Fig. 12.4) to annotate the underlying socket with the trace and span identifiers, which is performed via Netlink. From now on, any socket buffer (*i.e.*, packet) created for this socket will be automatically annotated with the trace and span identifiers. In line 3, the application sends an HTTPS request (*i.e.*, the monitored operation), reusing the same underlying socket as in line 2. At this stage, IOAM is already configured, and Pre-allocated Traces are present in the network traffic. In addition, for IPv6 packets sent from this socket, the trace and span identifiers are carried in the extended IOAM Pre-allocated Trace header. When the HTTPS request reaches the destination, the running IOAM agent reports IOAM Trace API objects to the IOAM collector via gRPC. The IOAM collector augments existing spans by attaching the corresponding IOAM data sent by the agent, and then forwards them to the backend collector for correlation with the original spans. In line 4, the application uses the CLT client library to remove annotations from the underlying socket immediately after receiving the response to the HTTPS request in line 3. As a result, the trace and span identifiers are no longer added to IPv6 packets sent from this socket. In line 5, the

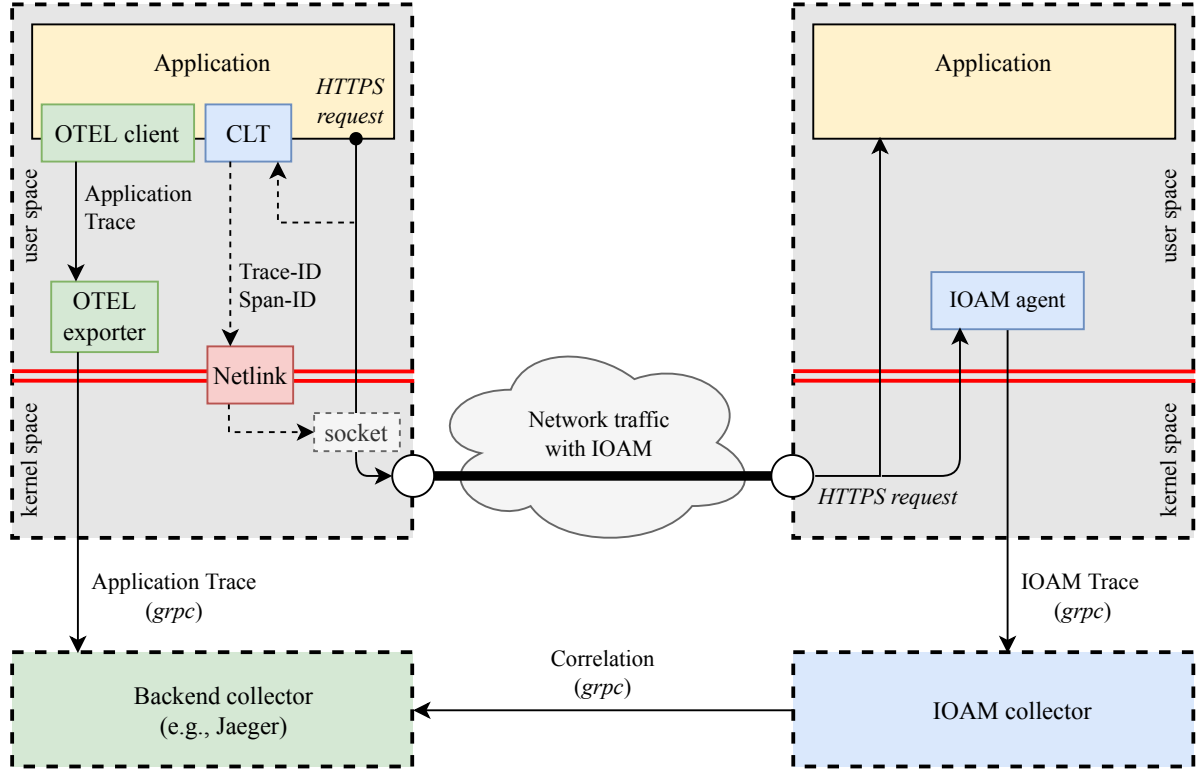


Figure 12.4: Cross-Layer Telemetry (CLT) architecture.

application uses the OpenTelemetry client library to stop the span, sending the corresponding APM trace to the OpenTelemetry exporter (the “OTEL exporter” green box in Fig. 12.4), which forwards it to the backend collector for storage. The backend database now contains the original spans augmented with IOAM data, providing both application and network telemetry in the same place.

12.2 Performance evaluation

Sec. 12.2.1 describes the methodology for evaluating each component of CLT separately (*i.e.*, the Netlink call, the IOAM agent, and the IOAM collector), while Sec. 12.2.2 discusses their performance results.

12.2.1 Methodology

To evaluate each component of CLT separately, we rely on TREX, which is already described in Chap. 3. Fig. 12.5 illustrates the straightforward testbed: one machine for TREX, and another one for the *Device Under Test* (DUT). Both are equipped with an *Intel XL710 2x40GB QSFP+* Network Interface Card (NIC), each connected port to port in order to close the loop (*i.e.*, TREX client and

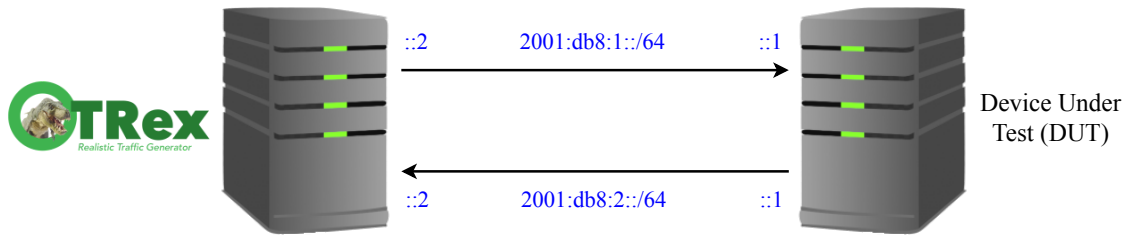


Figure 12.5: Testbed with TREX to evaluate each component of CLT separately.

server run on the same machine). This kind of topology provides an easy way to isolate a specific function on the DUT and evaluate it, *i.e.*, the Netlink call, the IOAM agent, and the IOAM collector in this case. The DUT has an *Intel Xeon e5-2630 v3 at 2.40GHz* Central Processing Unit (CPU), with 8 Cores, 16 Threads, and has 16GB RAM. It runs a Linux kernel version 5.17¹. 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. Overall, each experiment (*i.e.*, measurement) lasts 30 seconds and is run 20 times. The standard deviations are shown in the subsequent figures but are too close to be visible.

12.2.2 Results

As explained above, CLT is made of three distinct components: a Netlink call, the IOAM agent, and the IOAM collector. Their performance is evaluated in Sec. 12.2.2.1, Sec. 12.2.2.2, and Sec. 12.2.2.3 respectively. As for IOAM itself, it was evaluated in Chap. 9.

12.2.2.1 Netlink call

The results are based on one million calls for each operation (*i.e.*, enable or disable CLT) on the DUT. Fig. 12.6 shows the amount of time required to enable or disable CLT, with Go and Python. Without much surprise, it is way slower with Python because Python's interpreted runtime, higher-level abstractions, and dynamic data handling introduce significant overhead compared to Go's compiled, low-level, and closer-to-syscall execution path. With Go, the time spent for enabling or disabling CLT is quite identical and remains consistently low, with a global average of 75 microseconds. In case of performance concerns, the CLT client library in Go should be preferred. The overall impact is negligible and does not cause a performance bottleneck, allowing CLT to be enabled on demand.

¹At the time CLT was implemented, the most recent kernel version was 5.17. To date, this is the only kernel version for which a CLT patch is available.

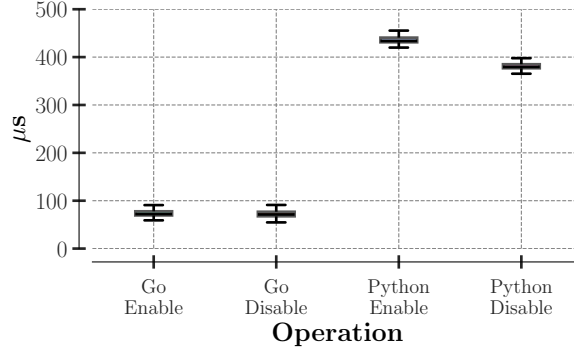


Figure 12.6: Time taken to enable or disable CLT with a Netlink call (Go vs. Python).

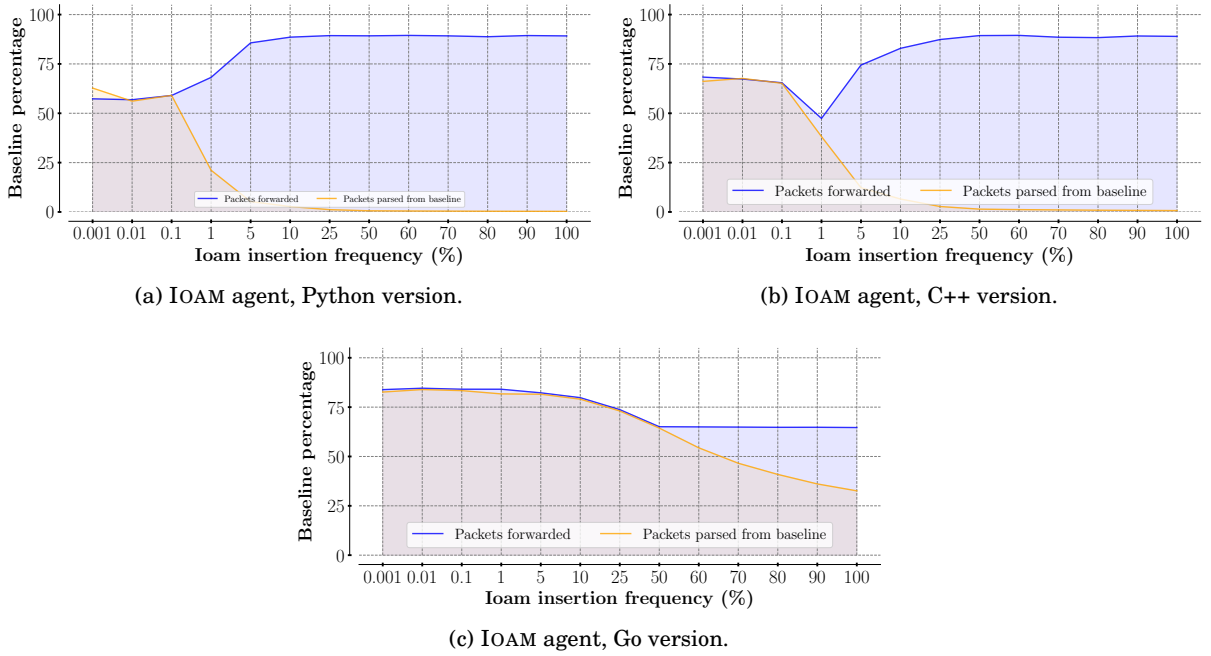


Figure 12.7: Performance of the IOAM agent.

12.2.2.2 IOAM agent

In this section, the term “baseline” refers to the maximum number of IPv6 packets per second that the DUT is able to forward. To evaluate the IOAM agent, we conduct two experiments: first, the impact of the IOAM agent on the number of packets per second that the DUT is able to forward (*i.e.*, baseline percentage); then, the number of packets per second that the IOAM agent is able to process (*i.e.*, parsing IOAM data and sending gRPC messages to the IOAM collector).

As shown in Fig. 12.7a (*i.e.*, IOAM agent in Python) and Fig. 12.7b (*i.e.*, IOAM agent in C++), the overall forwarding performance is considerably reduced. Both versions of the IOAM agent are

single-threaded and use a simple *AF_PACKET* socket to receive packets, which explains the poor performance. Furthermore, both figures show that parsing more packets in user space results in less forwarding by the kernel. As the proportion of packets carrying IOAM increases, the agent must perform additional parsing and processing, reducing the rate of successfully parsed packets (orange curve). In contrast, the kernel's forwarding performance (blue curve) improves because fewer packets are processed in user space, reducing contention on shared resources. In fact, even if the received traffic is steered to a single core (which is the case here) and packet parsing is performed by a user space application on another core, higher parsing loads can still degrade forwarding performance (*i.e.*, both due to CPU cycle contention and increased pressure on shared resources such as caches, memory bandwidth, and NIC queues).

Fig. 12.7c shows that the IOAM agent in Go performs better than in Python or C++. This version is multi-threaded, uses *PF_RING*, and benefits from the performance of Go, which explains the improvement. Packet processing in user space can handle up to 50% of network traffic containing IOAM, at the cost of a slight reduction of the baseline. Then, it starts to drop as forwarding takes over and remains stable, although also reduced. Since these measurements include gRPC messages sent to the IOAM collector, we also measured packet processing only as a way to check if one of the two is the bottleneck. In that case, there was a performance increase of about 20%, which unsurprisingly suggests that the packet processing part accounts for a larger portion of the processing time compared to sending gRPC messages, meaning it is the bottleneck. An improvement solution would be to use the *Zero Copy* mode of *PF_RING* to completely bypass the kernel network stack and thus increase performance.

Overall, the IOAM agent seems to be a bottleneck of CLT. Its impact on forwarding depends on the IOAM injection rate and may become problematic under heavy load. It means that some packets might not be processed, which would result in incomplete APM traces (*i.e.*, without network telemetry attached). On the other hand, these measurements represent a worst-case scenario.

12.2.2.3 IOAM collector

We evaluate the maximum number of gRPC messages the IOAM collector (written in Go) is able to process. For this experiment, we use a custom gRPC traffic generator to generate messages identical to those sent by the IOAM agent. Fig. 12.8 shows that the IOAM collector is able to process between 457,902 and 646,451 gRPC messages per second, depending on their size. The more IOAM data there is, the bigger the gRPC messages are and the less they are processed. Performance of the IOAM collector is also dependent on the backend tool (in this case, this is Jaeger). An optimization solution for both the IOAM agent and IOAM collector would be to have the IOAM agent send batches of messages, rather than dedicated messages. This would generate larger gRPC messages, but would reduce network load, allowing the IOAM agent to send more messages and the IOAM collector to process more messages.

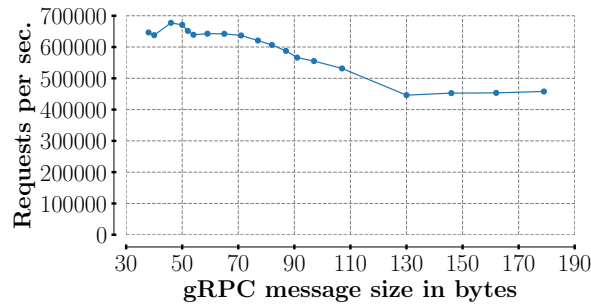


Figure 12.8: Number of gRPC messages the IOAM collector is able to process, depending on their size.

```
# tc qdisc add dev eth1 root netem delay 150ms
```

Listing 12.3: *tc* command to add a 150ms delay on an interface (*e.g.*, *eth1*).

12.3 Use case example

Let us picture a scenario where clients use a mobile application requiring authentication. The application sends an HTTPS request towards the corresponding remote API, with the username and password entered by the client. The receiving API entry point hides the business logic behind each request. In this case, a sub-request to authenticate the client is sent to a server. Each sub-request sent by the API entry point is traced by OpenTelemetry, and Jaeger is used as the backend. Suddenly, huge delays during the login process are reported by Jaeger. Indeed, login traces show longer than usual execution times. The operator investigates and, surprisingly, both the server and its local database appear healthy. Therefore, the decision is made to enable CLT as a way to quickly understand the problem. As the operator suspects that the culprit is network congestion somewhere, and to be proactive, the following IOAM data is required during IOAM configuration: node-id, ingress and egress interface identifiers, and egress queue depth.

Fig. 12.9 illustrates the scenario previously described. We use docker [67] to build the topology and docker-compose [68] to ease the configuration between each container. Each component is represented by a docker container. The application running on the API entry point uses the OpenTelemetry client library to monitor any critical part of the code, and uses the CLT client library to annotate the underlying socket (*i.e.*, kept open for all clients) with the respective trace and span identifiers. Still on the entry point, the OpenTelemetry exporter reports traces to a backend collector (*i.e.*, Jaeger collector in this example). An IOAM agent runs on the server and reports every IOAM trace to the IOAM collector. The operator uses Jaeger UI as a web interface to observe traces stored in the database. IPv6 is deployed between the entry point and the server. IOAM is enabled and configured on each node within the domain. The entry point is configured to insert an IOAM Pre-allocated Trace in packets when the destination is the server.

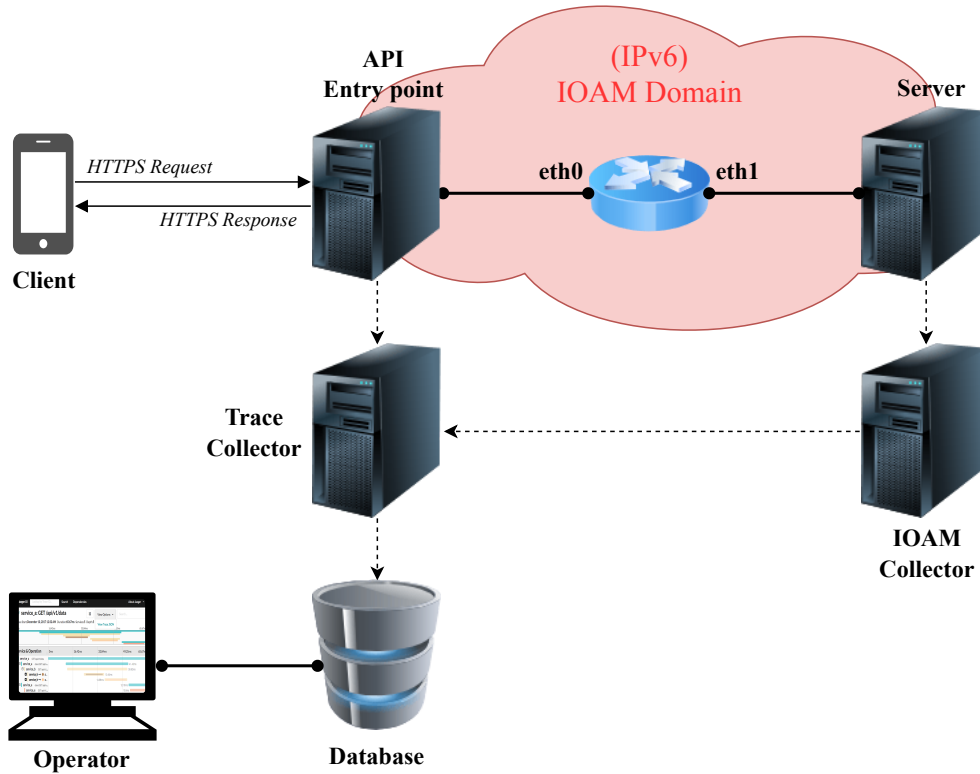


Figure 12.9: Example of a simplified use case with CLT.

To simulate a network issue, we introduce artificial delay to mimic a congestion on the router between the entry point and the server. Listing 12.3 shows the `tc` command used to add a delay of $150ms$ on the router interface towards the server, which means the RTT will suffer from a $300ms$ increase. In this experiment, we generate 200 HTTPS requests per second, over a period of four minutes. This time frame is divided in four slices (one minute each): the first minute represents a normal situation where everything runs fine. The congestion (additional $150ms$ delay) is introduced in the second minute. The third minute includes the problem investigation by enabling CLT. Finally, the last minute represents the come back to a normal situation, after the problem has been fixed by the operator. During the four minutes experiment, we measure the RTT for each HTTPS request.

Fig. 12.10 shows a screenshot of Jaeger with a span representing a login request that was randomly selected among all login requests during the third minute of the experiment (*i.e.*, with CLT enabled). The *ioam-span* is the augmented span attached to the original one. With that, the operator quickly detects that the egress queue of the router is increasing (see the red rectangle), indicating congestion. An action can then be applied to fix the problem, *e.g.*, by dynamically re-balancing traffic across egress queues, interfaces, or alternate routing paths to reduce congestion. Without CLT, the operator would have had much more difficulty performing root cause analysis.

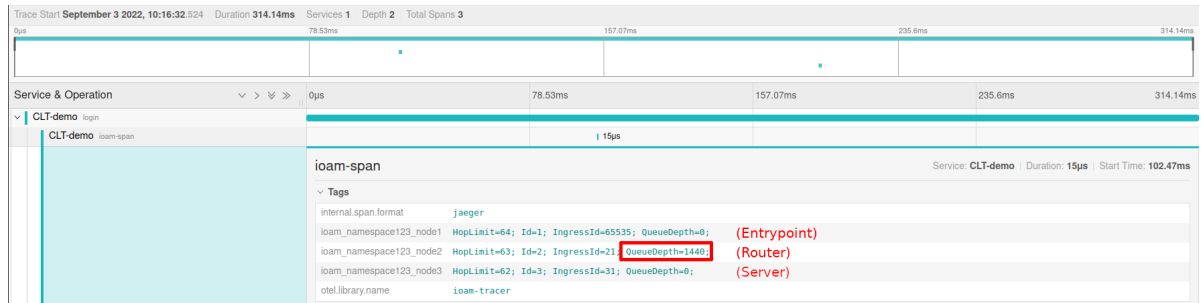


Figure 12.10: Example of an augmented span (with IOAM data) stored by Jaeger.

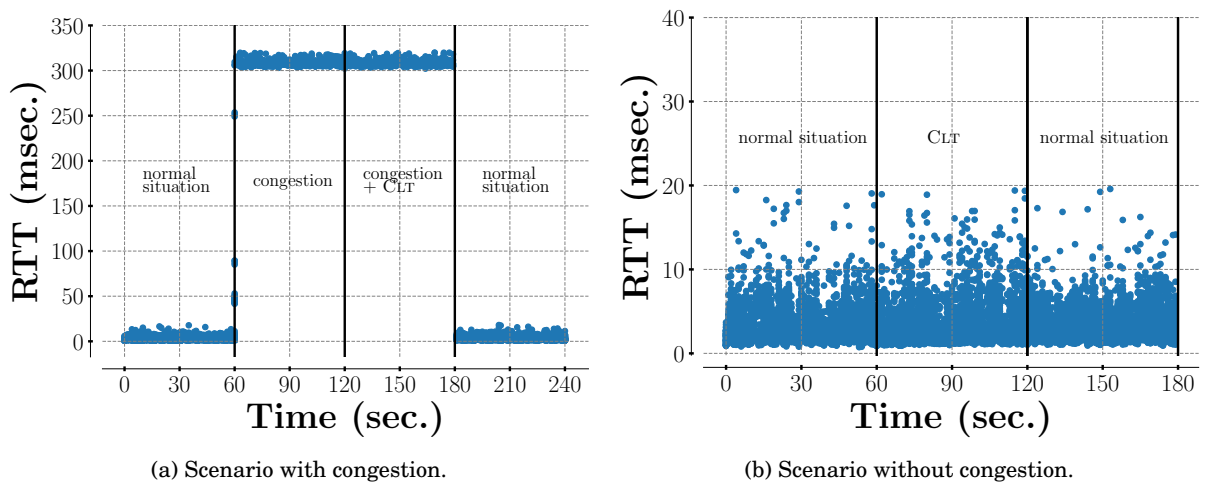


Figure 12.11: RTT measurement with and without CLT.

This experiment also allows to measure the overall impact of CLT (*e.g.*, on RTTs). Fig. 12.11a shows all measured RTTs during the four-minute experiment. During the second minute, one can clearly see that the RTT has increased by 300ms due to the simulated congestion. The key part is the third minute, during which CLT is enabled. Indeed, the distribution on the graph looks the same for both the second and third minute, demonstrating that CLT has little impact. In order to make sure that CLT is really effective, we also perform a similar experiment in three steps without congestion. The objective is to see how CLT behaves without congestion and so with more traffic. Fig. 12.11b shows all measured RTTs during the three-minute experiment. During the second minute (*i.e.*, CLT enabled without congestion), one can see that it only has a small impact on RTTs. Based on performance results in Sec. 12.2, that small impact is due to IOAM itself, the IOAM agent (especially), and the IOAM collector.

12.4 Discussion

Using a Netlink call for CLT offers a practical and modular approach, as it leverages an existing kernel-to-user communication mechanism without requiring changes to the system call interface or user space libraries. However, Netlink is not a lossless transport and may not be the perfect fit in this case: under heavy system load or high telemetry rates, messages may be dropped, potentially leading to incomplete or inconsistent observability data. In contrast, introducing a new or modified system call primitive could provide more reliable and efficient data delivery with tighter integration into the kernel's execution path. Yet, this approach comes with significant drawbacks: it would require modifications to the kernel, the C standard library (*libc*), socket APIs, and programming language runtimes to expose the new functionality, thereby complicating deployment and reducing portability. An intermediate option would be to use *LD_PRELOAD* [181] to intercept and extend existing system calls, allowing CLT to be integrated transparently into user applications without kernel or *libc* modifications. While this technique avoids invasive system-level changes, it still requires adaptations within programming language runtimes to ensure the intercepted behavior is correctly propagated. At the time of writing, a glimmer of hope has appeared with RFC 9622 [281], which specifies an abstract API for the transport layer. This API enables passing message contexts and associated properties alongside the data at the time of sending. Once it is supported in the Linux kernel and *libc*, and adopted by programming language runtimes, the current reliance on the Netlink call for passing identifiers to the kernel could be replaced by this standardized mechanism. Unfortunately, it may take a long time before this API is widely supported.

As previously mentioned, CLT relies on IOAM to kill two birds with one stone: (i) IOAM provides per-hop telemetry data and (ii) IOAM carries APM identifiers. The latter is achieved by extending the IOAM Pre-allocated Trace header, as described in Sec. 12.1.2. While it is perfectly fine for a proof-of-concept, it would require a more robust solution to go through standardization and be widely adopted. Indeed, the only advantage of this solution is to parse both IOAM data and APM identifiers in one-step (*i.e.*, they are gathered in the same place within packets). However, there are two disadvantages: first, it wouldn't be standardized as it is; second, there is a maximum of 244 precious bytes available for IOAM data with IPv6 (*i.e.*, not that much), which would only leave 220 bytes available because of the identifiers (vs. 188 bytes with integrity protection). To this end, we proposed an IETF draft [155] as a way to carry identifiers in IPv6 packets, more specifically in a Hop-by-Hop Options Header or Destination Options Header, which could be a perfect fit for CLT. As a result, the correlation between APM traces and network traffic would not be tied to IOAM anymore. One could use IOAM alone, or carry identifiers alone, or have both at the same time.

The advantages that CLT brings to operators are a considerable saving of time and a complete tool to detect network issues in applications, with the help of APM tools and distributed tracing. Overall, CLT offers a solution that is generic enough to integrate other backend alternatives

to Jaeger. Indeed, only the IOAM collector depends on the backend tool and can be quickly implemented for other backends, while the CLT client library and the IOAM agent are independent of the environment. Therefore, we do believe that CLT solves challenges related to *microservices* and distributed tracing in general, and is definitely useful for operators.

SUMMARY

The second part of this document aimed to answer the remaining Research Questions, which represent important topics for this research and help structure the investigation of advanced network observability. As a reminder, these Research Questions that will be answered in this chapter are:

- **RQ-2:** *“What is the impact of IOAM on network traffic?”*
- **RQ-3:** *“Can we ensure trust in IOAM data?”*
- **RQ-4:** *“Can we achieve unified observability that correlates network and application layers?”*

This chapter summarizes the points covered in the second part and, based on the data collected, concludes by answering each Research Question.

In Chap. 7, we provide the background to understand the difference between key terms such as telemetry, monitoring, and observability, and describe how they are achieved. In Chap. 8, we present In Situ Operations, Administration, and Maintenance (IOAM) in detail, covering terminology, header structures of Option-Types and how they work, deployment within IPv6 and related challenges, interesting use cases, and a comparison with similar protocols. In Chap. 9, we discuss existing IOAM implementations, with particular focus on the Linux kernel (*i.e.*, a key contribution to this research), and evaluate its performance. In Chap. 10, we describe the analysis and fix of a bug related to IOAM in the Linux kernel, and show our fix has no impact on performance. In Chap. 11, we explore multiple solutions to ensure integrity protection of IOAM data, evaluate each solution, and contribute to IETF standardization with the most suitable and effective solution. Finally, in Chap. 12, we demonstrate that cross-layer telemetry can be implemented in practice, although complex, by correlating IOAM data with application telemetry, and we evaluate performance of each component.

Based on what we have learned in the second part of this document, let us now conclude by answering the remaining Research Questions. Regarding **RQ-2** (“*What is the impact of IOAM on network traffic?*”), it can be answered as follows: IOAM has a measurable impact on network traffic at full line rate, which depends primarily on the IOAM injection rate and, to a lesser extent, on the size of the entire IOAM header (including data). It has been observed that, to minimize the impact on performance in general, operators should not exceed 1% of IOAM injection per CPU core (forwarding at line rate). All measurements and results are based on the Linux kernel implementation, which is a software implementation. Efficient hardware support would overcome these limitations, but TLVs are known to be hardware-unfriendly. However, this has proven feasible, as seen with SRv6 which has successful deployment [89, 189] in modern ASICs, but also due to the emergence of programmable data paths. Also, IOAM introduces an overhead that can significantly increase packet sizes, especially with Trace Option-Types. When injecting IOAM in packets, operators must pay attention to not exceed the MTU.

Regarding **RQ-3** (“*Can we ensure trust in IOAM data?*”), it can be answered positively. While IOAM by itself provides no integrity guarantees, it is possible to secure IOAM data through efficient per-hop cryptographic chaining techniques. Among the analyzed solutions, a cumulative GMAC-based protection, implemented and evaluated in this document, provides a robust balance between security, performance, and deployability in limited domains. However, this solution remains quite complex due to transit nodes potentially involved in protecting IOAM data integrity (*e.g.*, with Trace Option-Types).

Finally, **RQ-4** (“*Can we achieve unified observability that correlates network and application layers?*”) can be answered affirmatively. We implemented a prototype, called Cross-Layer Telemetry (CLT), which demonstrates that application telemetry can be correlated with network telemetry (*i.e.*, IOAM data) with a manageable impact and without disrupting packet processing pipelines. The difficulty of the proposed solution lies in the fact that it must integrate existing systems without requiring modifications to programming language runtimes, low-level APIs, or libraries. While additional optimization is possible, the approach is viable for advanced observability use cases in modern networks. In the long term, thanks to standardization and industry adoption, CLT will be even easier to deploy.

CONCLUSION

This research has been conducted in a context where the need for accurate, scalable, and low-overhead network observability has become increasingly critical for modern network operators, service providers, and large-scale distributed systems. As IPv6 continues to shape the future of Internet protocol design, IPv6 Extension Headers remain both an opportunity and a challenge. Although they offer a flexible mechanism for carrying optional per-packet information, their actual deployment has long been limited by concerns regarding performance, inconsistent device support, and security implications. In parallel, the IETF has introduced In Situ Operations, Administration, and Maintenance (IOAM) as a new OAM measurement technique that embeds telemetry directly inside user packets, enabling hop visibility that traditional active or passive tools cannot offer. Yet, IOAM itself is tightly coupled to IPv6 Extension Headers, particularly the Hop-by-Hop Options Header and Destination Options Header, and therefore inherits many of the operational uncertainties surrounding Extension Headers. These challenges, *i.e.*, understanding the real behavior of IPv6 Extension Headers and assessing the feasibility of IOAM as an IETF-standardized telemetry mechanism, define the central problem space addressed by this thesis, whose main ambition is to push network observability further by leveraging IPv6 Extension Headers and IOAM.

A first major contribution of this research revisits the practical behavior of IPv6 Extension Headers in both controlled and real-world environments. Historically, assumptions about the unreliability of Extension Headers have been widespread but insufficiently supported by modern measurements. By conducting extensive experiments across heterogeneous networks, including Internet-wide probing and controlled testbeds involving multiple hardware vendors, this thesis provides an updated and rigorous understanding of how Extension Headers propagate in practice. The results refine long-standing community assumptions, identify where Extension Headers remain viable, and clarify the operational conditions under which they can support measurement and telemetry functions.

A second major contribution of this research is the implementation of IOAM in the Linux kernel, integrated directly into the IPv6 data path. This implementation supports the IOAM Pre-allocated Trace Option-Type, providing the first open and RFC-compliant version in which IOAM can be evaluated under realistic forwarding conditions. The work required addressing architectural constraints in Linux, defining data structures and control-plane interfaces, and ensuring compliance with emerging standards. Building on this implementation, the thesis

presents a comprehensive performance evaluation of IOAM within the kernel networking stack. These results show how IOAM affects forwarding performance, where its scalability limits lie, and which operational profiles are feasible. This part of the thesis clarifies where IOAM introduces measurable overhead, where optimizations are effective, and which trade-offs shape its deployment prospects.

In parallel, this thesis contributes to the IETF standardization of secure in-band telemetry by defining a method for protecting IOAM data integrity. The proposed GMAC-based mechanism provides tamper-proof telemetry for IOAM’s single-domain trust model, while minimizing processing costs. This contribution fills a critical gap in IOAM and enables operators to rely on telemetry even under partially adversarial conditions.

Finally, this thesis introduces the concept of Cross-Layer Telemetry (CLT) as a means of bridging the persistent gap between network-level and application-level observability. While IOAM (*i.e.*, network telemetry) provides hop-level insights into forwarding behavior and application telemetry offers visibility into service performance and user experience, these two domains are rarely combined in practice. CLT addresses this limitation by enabling the correlation of in-band network telemetry with measurements traditionally collected at higher layers (*i.e.*, Layer 5 → 7). This thesis lays the groundwork for a full observability stack and shows how CLT can unify packet-level and service-level visibility. Such an integration makes it possible to diagnose complex performance issues that span both the network and the application stack. CLT therefore points toward a future in which in-band telemetry is not limited to network operators but becomes a shared observability foundation across layers.

Taken together, the contributions of this thesis advance the feasibility and understanding of IPv6-based observability, showing that IPv6 Extension Headers and IOAM can serve as powerful telemetry mechanisms. The results demonstrate that modern IPv6 observability can be significantly enhanced through the careful design and use of Extension Headers. While challenges remain in deployment and security, this thesis provides a solid and practical foundation upon which future improvements can be built.

Future Work and Research Directions

Future work following this thesis naturally divides into two broad areas: (*i*) IPv6 Extension Headers in general, and (*ii*) further evolution of IOAM and its integration with higher-layer observability frameworks as hardware capabilities and operational practices advance. The results presented in this thesis strengthen the technical basis for deploying Extension Headers and IOAM, but they also reveal several open questions where follow-up work is necessary.

A first research direction is the security aspects of IPv6 Extension Headers. While they have been extensively studied [1, 11, 100, 101, 109, 111–113, 124, 194, 285], the current specification still leaves room for ambiguity. In particular, RFC 8200 contains a formulation that permits

forwarding behavior which may be exploited for denial-of-service scenarios, notably in the context of the Destination Options Header. As part of ongoing IETF work, a planned follow-up to this thesis is to update the problematic and contradictory text in RFC 8200 (*i.e.*, which permits an infinite number of occurrences for each Extension Header) in order to remove these ambiguities and provide clearer operational guidance. This effort is also directly related to the current IETF work on updating RFC 8504 (*i.e.*, “*IPv6 Node Requirements*”), where it remains an open question whether the existing requirements (*i.e.*, limits on the number of options in a Hop-by-Hop Options Header or Destination Options Header) should remain unchanged or be tightened, or whether new limits should be adopted. For now, RFC 9673 (*i.e.*, *IPv6 Hop-by-Hop Options Processing Procedures*) does not appear to require modifications, although this assessment may evolve as discussions progress within the IETF. By mitigating this type of denial-of-service attack (*e.g.*, involving IPv6 packets with more than two Destination Options Header’s), concerns about the number of options within a Hop-by-Hop Options Header or a Destination Options Header are automatically reduced.

A second research direction is to continue evaluating modern routers across as many vendors as possible, in order to observe how hardware capabilities and limitations evolve with respect to IPv6 Extension Headers processing. Complementing these controlled tests, future work could involve repeating large-scale Internet measurements to assess whether improvements in Extension Headers processing are observable in operational networks over time.

In parallel, the eBPF tool developed in this thesis (*i.e.*, FISHNET), to inject Extension Headers in IPv6 traffic, can be improved. While the current prototype provides a flexible and efficient mechanism, several extensions would significantly broaden its applicability. These include improving the programmability of injection policies, enabling dynamic adaptation based on traffic characteristics or operator-defined constraints, and enhancing performance.

Another research direction is the security aspects of IOAM itself when used over IPv6. This thesis contributed a mechanism for protecting IOAM data integrity, but several open questions remain, including the interaction between IOAM domains and IPv6 security mechanisms, the impact of emerging encapsulation protocols on IOAM’s trust model, and the processing of new features (*e.g.*, the Loopback flag in Trace Option-Types, which can lead to amplification attacks).

Still regarding IOAM, its evolution and refinement as a telemetry mechanism can be investigated. Its future is particularly promising as programmable hardware (*e.g.*, P4, SmartNICs, modern NPUs, and emerging Linux offload capabilities) continue to gain widespread deployment. These may allow IOAM to operate with substantially reduced overhead while supporting richer processing pipelines than currently feasible in software. Alternative IOAM mechanisms also merit further exploration. For example, comparing the scalability and deployment practicality of the Pre-allocated Trace Option-Type versus the Direct Export (DEX) Option-Type, or investigating other hybrid approaches.

Finally, Cross-Layer Telemetry (CLT) offers a complementary avenue for future work. While

CONCLUSION

CLT has been introduced in this thesis to correlate network and application telemetry, its design can be further improved to simplify integration across heterogeneous systems without requiring extensive modifications. Research could focus on enhancing scalability and standardization, and on developing lightweight mechanisms that allow CLT to operate efficiently in large-scale distributed environments. By making CLT easier to deploy and more scalable, both network and application telemetry could become a unified, end-to-end observability framework for modern networks and services.

APPENDIX A

This appendix provides a brief overview of the IETF standardization process, its document categories, and its normative terminology.

The Internet Engineering Task Force (IETF) is an open international standards organization responsible for the development and maintenance of the protocols and architectures that constitute the Internet. It operates through topic-oriented working groups in areas such as routing, transport, security, and network operations. Participation is open to individuals and technical decisions are generally reached through the principle of “rough consensus and running code”, emphasizing both community agreement and practical implementation experience.

The IETF distinguishes between two main categories of technical documents: Internet-Drafts (I-Ds) and Requests for Comments (RFCs). An Internet-Draft is a temporary document used to present proposals, ongoing work, or discussion material within the community. Internet-Drafts are valid for a limited period of six months and may be updated, replaced, or withdrawn at any time. As such, they do not constitute official standards. In contrast, an RFC is a permanent publication issued after review and approval through the IETF process. Once published, an RFC is immutable and cannot be modified directly. Updates require the publication of a new RFC that updates or obsoletes the previous one, although minor issues may be reported through the RFC errata process. RFCs may have different statuses, including Informational, Experimental, Best Current Practice (BCP), or Standards Track documents, the latter progressing from Proposed Standard to Internet Standard.

IETF specifications rely on a precise normative terminology to express requirement levels unambiguously. This terminology is formally defined in RFC 2119 and later clarified by RFC 8174. In this context, the keywords *MUST*, *REQUIRED*, and *SHALL* denote an absolute requirement, whereas *MUST NOT* and *SHALL NOT* indicate an absolute prohibition. The keywords *SHOULD* and *RECOMMENDED* express a strong recommendation that may only be disregarded in justified circumstances, while *SHOULD NOT* and *NOT RECOMMENDED* indicate that a behavior is generally discouraged. Finally, the keywords *MAY* and *OPTIONAL* denote an optional feature or behavior left to the discretion of the implementer. This controlled vocabulary is essential to ensure interoperability and to avoid ambiguities in protocol specifications and implementations.

APPENDIX B

This appendix contains the latest version [165] of the entire IETF draft that specifies the YANG Data Model for the integrity protection of IOAM (version 08, at the time of writing), which is a companion document to the IETF draft presented in Appendix D. This IETF work has just been approved and is about to become an IETF RFC .

Workgroup: ippm
Internet-Draft: draft-ietf-ippm-ioam-integrity-yang-08
Published: 8 May 2026
Intended Status: Standards Track
Expires: 9 November 2026
Authors: J. Iurman T. Zhou
University of Liege Huawei

A YANG Data Model for In Situ Operations, Administration, and Maintenance (IOAM) Integrity-Protected Options

Abstract

In Situ Operations, Administration, and Maintenance (IOAM) is an example of an on-path hybrid measurement method to collect operational and telemetry information. The collected data may then be exported to systems that will use it to, e.g., monitor, measure, or (re)configure the network. Integrity Protection of In Situ Operations, Administration, and Maintenance (IOAM) Data Fields (RFC YYYY) defines IOAM Options with integrity protection, also called Integrity-Protected Options. This document defines a YANG data model for the management of these Integrity-Protected Options. The document also defines an IANA-maintained YANG module for IOAM Integrity Protection Methods.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 9 November 2026.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction

1.1. Editorial Note (to be removed by RFC Editor)

2. Conventions used in this document

2.1. Abbreviations

2.2. Terminology

2.3. Tree Diagrams

3. Design of the IOAM Integrity YANG Data Model

3.1. Overview

3.2. Integrity-Protected Pre-allocated Tracing Profile

3.3. Integrity-Protected Incremental Tracing Profile

3.4. Integrity-Protected Proof of Transit Profile

3.5. Integrity-Protected Edge-to-Edge Profile

4. IOAM Integrity YANG Module

5. Security Considerations

6. IANA Considerations

6.1. IANA-Maintained "iana-ioam-integrity-protection-methods" YANG Module

7. Acknowledgements

8. References

8.1. Normative References

8.2. Informative References

Appendix A. Initial Version of the IOAM Integrity Protection Methods IANA-Maintained YANG Module

Appendix B. Full tree of the IOAM Integrity YANG Data Model

Appendix C. Example of the Integrity-Protected Incremental Tracing Profile

[Appendix D. Example of the Integrity-Protected Pre-allocated Tracing Profile](#)

[Appendix E. Example of the Integrity-Protected Proof of Transit and Integrity-Protected Edge-to-Edge Profiles](#)

[Authors' Addresses](#)

1. Introduction

In Situ Operations, Administration, and Maintenance (IOAM) is an example of an on-path hybrid measurement method [RFC7799] to collect operational and telemetry information. The collected data may then be exported to systems that will use it to, e.g., monitor, measure, or (re)configure the network. [I-D.ietf-ippm-ioam-data-integrity] defines IOAM Options with integrity protection, also called Integrity-Protected Options.

This document defines a YANG data model [RFC7950] for the management of these Integrity-Protected Options. This YANG data model supports four IOAM Integrity-Protected Options, which are as follows:

- Integrity-Protected Incremental Trace-Option (Section 4.1 of [I-D.ietf-ippm-ioam-data-integrity])
- Integrity-Protected Pre-allocated Trace-Option (Section 4.1 of [I-D.ietf-ippm-ioam-data-integrity])
- Integrity-Protected Proof of Transit (POT) Option (Section 4.2 of [I-D.ietf-ippm-ioam-data-integrity])
- Integrity-Protected Edge-to-Edge (E2E) Option (Section 4.3 of [I-D.ietf-ippm-ioam-data-integrity])

[I-D.ietf-ippm-ioam-data-integrity], Section 4, states that Direct Export (DEX) [RFC9326] is outside the scope of the Integrity Protection Method defined in that document, as DEX does not carry IOAM-Data-Fields. Consequently, no Integrity-Protected Option is defined for DEX, and it is not included in this YANG data model.

The document also defines an IANA-maintained YANG module for IOAM Integrity Protection Methods.

1.1. Editorial Note (to be removed by RFC Editor)

Note to the RFC Editor: this section is to be removed prior to publication.

This document contains placeholder values that need to be replaced with finalized values at the time of publication. This note summarizes all of the substitutions that are needed. No other RFC Editor instructions are specified elsewhere in this document, except in [Appendix A](#).

Please apply the following replacements:

- XXXX --> the assigned RFC number for this document
- YYYY --> the assigned RFC number for [[I-D.ietf-ippm-ioam-data-integrity](#)]
- 2026-05-08 --> the actual date of the publication of this document

2. Conventions used in this document

2.1. Abbreviations

Abbreviations used in this document:

OAM: Operations, Administration, and Maintenance

IOAM: In Situ OAM

POT: Proof of Transit

E2E: Edge to Edge

2.2. Terminology

The following terms are defined in [[RFC7950](#)] and are used in this specification:

- augment
- data model
- data node

The terminology for describing YANG data models is found in [[RFC7950](#)].

2.3. Tree Diagrams

Tree diagrams used in this document follow the notation defined in [[RFC8340](#)].

3. Design of the IOAM Integrity YANG Data Model

3.1. Overview

The IOAM Integrity model is organized as a list of profiles, as shown in [Figure 1](#). Full tree is provided in [Appendix B](#). In this model, the "int" prefix refers to "INTEGRity protection".

```

module: ietf-ioam-integrity

augment /ioam:ioam/ioam:profiles/ioam:profile:
  +--rw int-incremental-tracing-profile! {int-incremental-trace}?
  |   ...
  +--rw int-preallocated-tracing-profile! {int-preallocated-trace}?
  |   ...
  +--rw int-pot-profile! {int-proof-of-transit}?
  |   ...
  +--rw int-e2e-profile! {int-edge-to-edge}?

```

Figure 1: Overview of the IOAM Integrity model

This document defines augmentations to the "ietf-ioam" YANG module [RFC9617] by adding integrity-related profiles. Each profile is associated with one flow and the corresponding IOAM information. These integrity-related profiles are indicated by four defined features, i.e., "int-incremental-trace", "int-preallocated-trace", "int-proof-of-transit", and "int-edge-to-edge". The structures of the new profiles follow what is defined in [RFC9617], but for distinct purposes.

3.2. Integrity-Protected Pre-allocated Tracing Profile

As illustrated in Figure 2, the "int-preallocated-tracing-profile" container provides the detailed information for the pre-allocated tracing data with integrity protection. This information has the same structure as the Pre-allocated Tracing Profile (Section 3.2 of [RFC9617]), but has the following additional data node:

int-method: indicates which Integrity Protection Method is used, as defined in the "IOAM Integrity Protection Methods" IANA registry [IANA-IOAM]. It is only defined at the encapsulating node because the Integrity Protection Method is selected and initialized when IOAM data is encapsulated.

```

+--rw int-preallocated-tracing-profile! {int-preallocated-trace}?
  +--rw node-action?   ioam-node-action
  +--rw trace-types
  |   +--rw use-namespace?   ioam-namespace
  |   +--rw trace-type*      ioam-trace-type
  +--rw max-length?      uint32
  +--rw int-method        iana-ioam-ipm:method-id

```

Figure 2: Integrity-Protected Pre-allocated Tracing Profile

An example of the Integrity-Protected Pre-allocated Tracing Profile with data is provided in [Appendix D](#).

3.3. Integrity-Protected Incremental Tracing Profile

As illustrated in [Figure 3](#), the "int-incremental-tracing-profile" container provides the detailed information for the incremental tracing data with integrity protection. This information has the same structure as the Incremental Tracing Profile (Section 3.3 of [\[RFC9617\]](#)), but has the following additional data node:

int-method: indicates which Integrity Protection Method is used, as defined in the "IOAM Integrity Protection Methods" IANA registry [\[IANA-IOAM\]](#). It is only defined at the encapsulating node because the Integrity Protection Method is selected and initialized when IOAM data is encapsulated.

```
+--rw int-incremental-tracing-profile! {int-incremental-trace}?
  +--rw node-action?      ioam-node-action
  +--rw trace-types
  |   +--rw use-namespace?  ioam-namespace
  |   +--rw trace-type*     ioam-trace-type
  +--rw max-length?        uint32
  +--rw int-method         iana-ioam-ipm:method-id
```

Figure 3: Integrity-Protected Incremental Tracing Profile

An example of the Integrity-Protected Incremental Tracing Profile with data is provided in [Appendix C](#).

3.4. Integrity-Protected Proof of Transit Profile

As illustrated in [Figure 4](#), the "int-pot-profile" container is intended to provide the detailed information for the proof of transit data with integrity protection. This information has the same structure as the Proof of Transit Profile (Section 3.5 of [\[RFC9617\]](#)), but has the following additional data nodes:

node-action: imported from the "ietf-ioam" YANG module [\[RFC9617\]](#) with the same definition.

int-method: indicates which Integrity Protection Method is used, as defined in the "IOAM Integrity Protection Methods" IANA registry [\[IANA-IOAM\]](#). It is only defined at the encapsulating node because the Integrity Protection Method is selected and initialized when IOAM data is encapsulated.

```

+--rw int-pot-profile! {int-proof-of-transit}?
  +--rw use-namespace?   ioam:ioam-namespace
  +--rw pot-type?        ioam:ioam-pot-type
  +--rw node-action?     ioam:ioam-node-action
  +--rw int-method       iana-ioam-ipm:method-id

```

Figure 4: Integrity-Protected Proof of Transit Profile

An example of the Integrity-Protected Proof of Transit Profile with data is provided in [Appendix E](#).

3.5. Integrity-Protected Edge-to-Edge Profile

As illustrated in [Figure 5](#), the "int-e2e-profile" container provides the detailed information for the edge-to-edge data with integrity protection. This information has the same structure as the Edge-to-Edge Profile (Section 3.6 of [RFC9617](#)), but has the following additional data node:

int-method: indicates which Integrity Protection Method is used, as defined in the "IOAM Integrity Protection Methods" IANA registry [[IANA-IOAM](#)]. It is only defined at the encapsulating node because the Integrity Protection Method is selected and initialized when IOAM data is encapsulated.

```

+--rw int-e2e-profile! {int-edge-to-edge}?
  +--rw node-action?   ioam-node-action
  +--rw e2e-types
  | +--rw use-namespace?   ioam-namespace
  | +--rw e2e-type*        ioam-e2e-type
  +--rw int-method       iana-ioam-ipm:method-id

```

Figure 5: Integrity-Protected Edge-to-Edge Profile

An example of the Integrity-Protected Edge-to-Edge Profile with data is provided in [Appendix E](#).

4. IOAM Integrity YANG Module

The "ietf-ioam-integrity" module defined in this document imports the "ietf-ioam" module defined in [RFC9617](#). This document also references [I-D.ietf-ippm-ioam-data-integrity](#).

```
<CODE BEGINS> file "ietf-ioam-integrity@2026-05-08.yang"

module ietf-ioam-integrity {
  yang-version 1.1;
  namespace "urn:ietf:params:xml:ns:yang:ietf-ioam-integrity";
  prefix ioam-int;

  import ietf-ioam {
    prefix ioam;
    reference
      "RFC 9617: A YANG Data Model for In Situ Operations,
      Administration, and Maintenance (IOAM)";
  }

  import iana-ioam-integrity-protection-methods {
    prefix iana-ioam-ipm;
    reference
      "RFC XXXX: A YANG Data Model for In Situ Operations,
      Administration, and Maintenance (IOAM)
      Integrity-Protected Options";
  }

  organization
    "IETF IPPM (IP Performance Measurement) Working Group";

  contact
    "WG Web:  <https://datatracker.ietf.org/wg/ippm>
    WG List:  IPPM <mailto:ippm@ietf.org>

    Editor:    Tianran Zhou
               <mailto:zhoutianran@huawei.com>
    Editor:    Justin Iurman
               <mailto:justin.iurman@uliege.be>";

  description
    "This YANG module specifies a vendor-independent data model for
    In Situ Operations, Administration, and Maintenance (IOAM)
    Integrity-Protected Options.

    Copyright (c) 2026 IETF Trust and the persons identified as
    authors of the code.  All rights reserved.

    Redistribution and use in source and binary forms, with or
    without modification, is permitted pursuant to, and subject
    to the license terms contained in, the Revised BSD License
    set forth in Section 4.c of the IETF Trust's Legal Provisions
    Relating to IETF Documents
    (https://trustee.ietf.org/license-info).

    All revisions of IETF and IANA published modules can be found
    at the YANG Parameters registry group
    (https://www.iana.org/assignments/yang-parameters).

    This version of this YANG module is part of RFC XXXX; see the
    RFC itself for full legal notices.
```

Operational Considerations:

- This model does not treat as mutually exclusive an IOAM Option and its integrity protected equivalent (i.e., an IOAM Integrity-Protected Option). For example, an implementation may support the simultaneous configuration of an IOAM-Namespace with the Pre-allocated Trace Option and another IOAM-Namespace with the Integrity-Protected Pre-allocated Trace Option. Therefore, the model does not impose constraints that would prevent such use cases.";

```
revision 2026-05-08 {
  description
    "Initial version.";
  reference
    "RFC XXXX: A YANG Data Model for In Situ Operations,
      Administration, and Maintenance (IOAM)
      Integrity-Protected Options";
}

/*
 * FEATURES
 */

feature int-incremental-trace {
  description
    "This feature indicates that the Integrity-Protected
      Incremental Trace-Option is supported.";
  reference
    "RFC YYYY: Integrity Protection of In Situ Operations,
      Administration, and Maintenance (IOAM)
      Data Fields, Section 4.1";
}

feature int-preallocated-trace {
  description
    "This feature indicates that the Integrity-Protected
      Pre-allocated Trace-Option is supported.";
  reference
    "RFC YYYY: Integrity Protection of In Situ Operations,
      Administration, and Maintenance (IOAM)
      Data Fields, Section 4.1";
}

feature int-proof-of-transit {
  description
    "This feature indicates that the Integrity-Protected Proof of
      Transit Option is supported.";
  reference
    "RFC YYYY: Integrity Protection of In Situ Operations,
      Administration, and Maintenance (IOAM)
      Data Fields, Section 4.2";
}

feature int-edge-to-edge {
  description
    "This feature indicates that the Integrity-Protected
      Edge-to-Edge Option is supported.";
```

```
reference
  "RFC YYYY: Integrity Protection of In Situ Operations,
    Administration, and Maintenance (IOAM)
    Data Fields, Section 4.3";
}

/*
 * GROUP DEFINITIONS
 */

grouping int-method-grouping {
  description
    "A grouping for Integrity Protection Methods.";
  leaf int-method {
    when "derived-from-or-self(../node-action,
      'ioam:action-encapsulate')";
    type iana-ioam-ipm:method-id;
    mandatory true;
    description
      "This object indicates the Integrity Protection Method for
        this profile. 'int-method' is only defined at the
        encapsulating node.";
  }
}

/*
 * DATA NODES
 */

augment "/ioam:ioam/ioam:profiles/ioam:profile" {
  description
    "This augmentation adds 4 profiles for the Integrity-Protected
    Options.";
  container int-incremental-tracing-profile {
    if-feature "int-incremental-trace";
    presence "Enables the Integrity-Protected
      Incremental Trace-Option.";
    description
      "This container describes the profile for the
      Integrity-Protected Incremental Trace-Option.";
    uses ioam:ioam-incremental-tracing-profile;
    uses int-method-grouping;
  }
  container int-preallocated-tracing-profile {
    if-feature "int-preallocated-trace";
    presence "Enables the Integrity-Protected
      Pre-allocated Trace-Option.";
    description
      "This container describes the profile for the
      Integrity-Protected Pre-allocated Trace-Option.";
    uses ioam:ioam-preallocated-tracing-profile;
    uses int-method-grouping;
  }
  container int-pot-profile {
    if-feature "int-proof-of-transit";
    presence "Enables the Integrity-Protected
      Proof of Transit Option.";
    description
```

```
    "This container describes the profile for the
    Integrity-Protected Proof of Transit Option.";
  leaf use-namespace {
    type ioam:ioam-namespace;
    default "ioam:default-namespace";
    description
      "This object indicates the namespace used for the
      POT types.";
  }
  leaf pot-type {
    type ioam:ioam-pot-type;
    description
      "Type of a particular POT variant that specifies
      the POT data that is included.";
  }
  leaf node-action {
    type ioam:ioam-node-action;
    default "ioam:action-transit";
    description
      "This object indicates the action the node needs to
      take, e.g., encapsulation.";
  }
  uses int-method-grouping;
}
container int-e2e-profile {
  if-feature "int-edge-to-edge";
  presence "Enables the Integrity-Protected
  Edge-to-Edge Option.";
  description
    "This container describes the profile for the
    Integrity-Protected Edge-to-Edge Option.";
  uses ioam:ioam-e2e-profile;
  uses int-method-grouping;
}
}
```

<CODE ENDS>

5. Security Considerations

This section is modeled after the template described in Section 3.7.1 of [RFC9907].

The "ietf-ioam-integrity" YANG module defines a data model that is designed to be accessed via YANG-based management protocols, such as the Network Configuration Protocol (NETCONF) [RFC6241] and RESTCONF [RFC8040]. These YANG-based management protocols (1) have to use a secure transport layer (e.g., Secure Shell (SSH) [RFC4252], TLS [I-D.ietf-tls-rfc8446bis], and QUIC [RFC9000]) and (2) have to use mutual authentication.

The Network Configuration Access Control Model (NACM) [RFC8341] provides the means to restrict access for particular NETCONF or RESTCONF users to a preconfigured subset of all available NETCONF or RESTCONF protocol operations and content.

There are a number of data nodes defined in this YANG module that are writable/creatable/deletable (i.e., "config true", which is the default). All writable data nodes are likely to be sensitive or vulnerable in some network environments. Write operations (e.g., edit-config) and delete operations to these data nodes without proper protection or authentication can have a negative effect on network operations. The following subtrees and data nodes have particular sensitivities/vulnerabilities:

/ioam:ioam/ioam:profiles/ioam:profile: The entries in the "profile" list include the whole IOAM profile configurations. Unexpected changes to these entries could lead to incorrect IOAM behavior for the corresponding flows. Consequently, such changes would impact performance monitoring, data analytics, and associated interactions with network services. Also, unauthorized access to integrity-related parameters may impact the integrity protection service, thus preventing the interpretation and validation of IOAM data.

Some of the readable data nodes in this YANG module may be considered sensitive or vulnerable in some network environments. It is thus important to control read access (e.g., via get, get-config, or notification) to these data nodes. Specifically, the following subtrees and data nodes have particular sensitivities/vulnerabilities:

/ioam:ioam/ioam:profiles/ioam:profile: The information contained in this subtree might reveal information about the services deployed for customers. For instance, a customer might be given access to monitor the status of their services. In this scenario, the customer's access should be restricted to nodes representing their services so as not to divulge information about the underlying network structure or services.

This YANG module uses groupings from other YANG modules that define nodes that may be considered sensitive or vulnerable in network environments. Refer to the Security Considerations of [RFC9617] for information as to which nodes may be considered sensitive or vulnerable in network environments.

6. IANA Considerations

IANA is requested to register the following URIs in the "ns" registry within the "IETF XML Registry" group [RFC3688]:

URI: urn:ietf:params:xml:ns:yang:ietf-ioam-integrity

Registrant Contact: The IESG.

XML: N/A; the requested URI is an XML namespace.

URI: urn:ietf:params:xml:ns:yang:iana-ioam-integrity-protection-methods

Registrant Contact: The IESG.

XML: N/A; the requested URI is an XML namespace.

IANA is requested to register the following YANG modules in the "YANG Module Names" registry [[RFC6020](#)] within the "YANG Parameters" registry group:

Name: ietf-ioam-integrity
Maintained by IANA?: N
Namespace: urn:ietf:params:xml:ns:yang:ietf-ioam-integrity
Prefix: ioam-int
Reference: RFC XXXX

Name: iana-ioam-integrity-protection-methods
Maintained by IANA?: Y
Namespace: urn:ietf:params:xml:ns:yang:iana-ioam-integrity-protection-methods
Prefix: iana-ioam-ipm
Reference: RFC XXXX

6.1. IANA-Maintained "iana-ioam-integrity-protection-methods" YANG Module

This document defines the initial version of the IANA-maintained "iana-ioam-integrity-protection-methods" YANG module. The most recent version of the YANG module is available from the "YANG Parameters" registry group [[IANA-YANG-PARAMETERS](#)].

IANA is requested to add this note to the registry:

New values must not be directly added to the "iana-ioam-integrity-protection-methods" YANG module. They must instead be added to the "IOAM Integrity Protection Methods" registry.

When a value is added to the "IOAM Integrity Protection Methods" registry, a new "enum" statement must be added to the "iana-ioam-integrity-protection-methods" YANG module. The "enum" statement, and sub-statements thereof, should be defined:

"enum": Prefix "method-" to the decimal value of the ID from the registry.
"value": Contains the decimal value of the IANA-assigned ID value.
"status": Is included only if a registration has been deprecated or obsoleted. IANA "deprecated" maps to YANG status "deprecated", and IANA "obsolete" maps to YANG status "obsolete".
"description": Replicates the description from the registry.
"reference": Replicates the reference(s) from the registry. References to documents should also include titles.

IANA is requested to add this note to [[reference-to-the-iana-foo-registry](#)]:

When this registry is modified, the YANG module "iana-ioam-integrity-protection-methods" [IANA_FOO_URL] must be updated as defined in RFC XXXX.

7. Acknowledgements

The authors would like to thank Alex Huang Feng, Will Hawkins, Mohamed Boucadair, Corey Bonnell, Satoru Matsushima, Roni Even, and Qiufang Ma for their valuable feedback.

8. References

8.1. Normative References

- [I-D.ietf-ippm-ioam-data-integrity] Brockners, F., Bhandari, S., Mizrahi, T., and J. Iurman, "Integrity Protection of In Situ Operations, Administration, and Maintenance (IOAM) Data Fields", Work in Progress, Internet-Draft, draft-ietf-ippm-ioam-data-integrity-19, 11 May 2026, <<https://datatracker.ietf.org/doc/html/draft-ietf-ippm-ioam-data-integrity-19>>.
- [IANA-IOAM] "In Situ OAM (IOAM)", n.d., <<https://www.iana.org/assignments/ioam/ioam.xhtml>>.
- [RFC3688] Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688, DOI 10.17487/RFC3688, January 2004, <<https://www.rfc-editor.org/info/rfc3688>>.
- [RFC6020] Bjorklund, M., Ed., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, DOI 10.17487/RFC6020, October 2010, <<https://www.rfc-editor.org/info/rfc6020>>.
- [RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", RFC 7950, DOI 10.17487/RFC7950, August 2016, <<https://www.rfc-editor.org/info/rfc7950>>.
- [RFC8341] Bierman, A. and M. Bjorklund, "Network Configuration Access Control Model", STD 91, RFC 8341, DOI 10.17487/RFC8341, March 2018, <<https://www.rfc-editor.org/info/rfc8341>>.
- [RFC9617] Zhou, T., Ed., Guichard, J., Brockners, F., and S. Raghavan, "A YANG Data Model for In Situ Operations, Administration, and Maintenance (IOAM)", RFC 9617, DOI 10.17487/RFC9617, August 2024, <<https://www.rfc-editor.org/info/rfc9617>>.

8.2. Informative References

- [I-D.ietf-tls-rfc8446bis] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-rfc8446bis-14, 13 September 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-tls-rfc8446bis-14>>.

[IANA-YANG-PARAMETERS] "YANG Parameters", n.d., <<https://www.iana.org/assignments/yang-parameters>>.

[RFC4252] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Authentication Protocol", RFC 4252, DOI 10.17487/RFC4252, January 2006, <<https://www.rfc-editor.org/info/rfc4252>>.

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011, <<https://www.rfc-editor.org/info/rfc6241>>.

[RFC7799] Morton, A., "Active and Passive Metrics and Methods (with Hybrid Types In-Between)", RFC 7799, DOI 10.17487/RFC7799, May 2016, <<https://www.rfc-editor.org/info/rfc7799>>.

[RFC7951] Lhotka, L., "JSON Encoding of Data Modeled with YANG", RFC 7951, DOI 10.17487/RFC7951, August 2016, <<https://www.rfc-editor.org/info/rfc7951>>.

[RFC8040] Bierman, A., Bjorklund, M., and K. Watsen, "RESTCONF Protocol", RFC 8040, DOI 10.17487/RFC8040, January 2017, <<https://www.rfc-editor.org/info/rfc8040>>.

[RFC8340] Bjorklund, M. and L. Berger, Ed., "YANG Tree Diagrams", BCP 215, RFC 8340, DOI 10.17487/RFC8340, March 2018, <<https://www.rfc-editor.org/info/rfc8340>>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <<https://www.rfc-editor.org/info/rfc9000>>.

[RFC9326] Song, H., Gafni, B., Brockners, F., Bhandari, S., and T. Mizrahi, "In Situ Operations, Administration, and Maintenance (IOAM) Direct Exporting", RFC 9326, DOI 10.17487/RFC9326, November 2022, <<https://www.rfc-editor.org/info/rfc9326>>.

[RFC9907] Bierman, A., Boucadair, M., Ed., and Q. Wu, "Guidelines for Authors and Reviewers of Documents Containing YANG Data Models", BCP 216, RFC 9907, DOI 10.17487/RFC9907, March 2026, <<https://www.rfc-editor.org/info/rfc9907>>.

Appendix A. Initial Version of the IOAM Integrity Protection Methods IANA-Maintained YANG Module

RFC Ed.: please remove this section.

```
<CODE BEGINS> file "iana-ioam-integrity-protection-methods@2026-05-08.yang"

module iana-ioam-integrity-protection-methods {
  yang-version 1.1;
  namespace "urn:ietf:params:xml:ns:yang:iana-ioam-integrity-protection-
methods";
  prefix iana-ioam-ipm;

  organization
    "Internet Assigned Numbers Authority (IANA)";

  contact
    "Internet Assigned Numbers Authority

    ICANN
    12025 Waterfront Drive, Suite 300
    Los Angeles, CA 90094

    Tel: +1 310 301 5800

    <mailto:iana@iana.org>";

  description
    "This YANG module is maintained by IANA and reflects
    the 'IOAM Integrity Protection Methods' registry.

    Copyright (c) 2026 IETF Trust and the persons
    identified as authors of the code. All rights reserved.

    Redistribution and use in source and binary forms, with or
    without modification, is permitted pursuant to, and subject to
    the license terms contained in, the Revised BSD License set
    forth in Section 4.c of the IETF Trust's Legal Provisions
    Relating to IETF Documents
    (https://trustee.ietf.org/license-info).

    All revisions of IETF and IANA published modules can be found
    at the YANG Parameters registry group
    (https://www.iana.org/assignments/yang-parameters).

    The initial version of this YANG module is part of RFC XXXX;
    see the RFC itself for full legal notices.

    // RFC Ed.: replace the IANA_F00_URL and remove this note

    The latest version of this YANG module is available at
    <IANA_F00_URL>.";

  reference
    "IOAM Integrity Protection Methods
    (https://www.iana.org/assignments/ioam/ioam.xhtml)";

  revision 2026-05-08 {
    description
      "Initial version.";
    reference
```

```
    "URL of the published initial version of the module

// RFC Ed.: replace above with the URL of the module
//          and remove this note

    RFC XXXX: A YANG Data Model for In Situ Operations,
              Administration, and Maintenance (IOAM)
              Integrity-Protected Options";
}

/*
 * Type definitions
 */

typedef method-id {
  type enumeration {
    enum method-0 {
      value 0;
      description
        "AES-GMAC, 16-octet (full) Authentication Tag,
        12-octet Initialization Vector.";
      reference
        "RFC YYYY: Integrity Protection of In Situ Operations,
        Administration, and Maintenance (IOAM)
        Data Fields, Section 5";
    }
  }
  description
    "Identifier for an IOAM Integrity Protection Method,
    matching the 'IOAM Integrity Protection Methods'
    IANA registry.";
  reference
    "https://www.iana.org/assignments/ioam/ioam.xhtml";
}
}

<CODE ENDS>
```

Appendix B. Full tree of the IOAM Integrity YANG Data Model

Figure 6 illustrates the full tree of the IOAM Integrity YANG Data Model.

```

module: ietf-ioam-integrity

augment /ioam:ioam/ioam:profiles/ioam:profile:
  +--rw int-incremental-tracing-profile! {int-incremental-trace}?
  |   +--rw node-action?    ioam-node-action
  |   +--rw trace-types
  |   |   +--rw use-namespace?    ioam-namespace
  |   |   +--rw trace-type*       ioam-trace-type
  |   +--rw max-length?    uint32
  |   +--rw int-method      iana-ioam-ipm:method-id
  +--rw int-preallocated-tracing-profile! {int-preallocated-trace}?
  |   +--rw node-action?    ioam-node-action
  |   +--rw trace-types
  |   |   +--rw use-namespace?    ioam-namespace
  |   |   +--rw trace-type*       ioam-trace-type
  |   +--rw max-length?    uint32
  |   +--rw int-method      iana-ioam-ipm:method-id
  +--rw int-pot-profile! {int-proof-of-transit}?
  |   +--rw use-namespace?    ioam:ioam-namespace
  |   +--rw pot-type?         ioam:ioam-pot-type
  |   +--rw node-action?      ioam:ioam-node-action
  |   +--rw int-method        iana-ioam-ipm:method-id
  +--rw int-e2e-profile! {int-edge-to-edge}?
  |   +--rw node-action?      ioam-node-action
  |   +--rw e2e-types
  |   |   +--rw use-namespace?    ioam-namespace
  |   |   +--rw e2e-type*         ioam-e2e-type
  |   +--rw int-method        iana-ioam-ipm:method-id

grouping int-method-grouping:
  +-- int-method      iana-ioam-ipm:method-id

```

Figure 6: Full tree of the IOAM Integrity YANG Data Model

Appendix C. Example of the Integrity-Protected Incremental Tracing Profile

A JSON [RFC7951] example of the Integrity-Protected Incremental Tracing Profile is depicted in Figure 7. This configuration is received by an IOAM ingress node. This node encapsulates the IOAM data in the IPv6 Hop-by-Hop option header. The Integrity Protection Method to be used is method 0. The trace type indicates that each on-path node needs to capture the transit delay and add the data to the IOAM node data list. The incremental tracing data space is variable; however, the node data list must not exceed 512 bytes.

```
{
  "ietf-ioam:ioam": {
    "admin-config": {
      "enabled": true
    },
    "profiles": {
      "profile": [
        {
          "profile-name": "ietf-test-profile",
          "protocol-type": "ietf-ioam:ipv6",
          "ietf-ioam-integrity:int-incremental-tracing-profile": {
            "node-action": "ietf-ioam:action-encapsulate",
            "trace-types": {
              "use-namespace": "ietf-ioam:default-namespace",
              "trace-type": [
                "ietf-ioam:trace-transit-delay"
              ]
            },
            "max-length": 512,
            "int-method": "method-0"
          }
        ]
      ]
    }
  }
}
```

Figure 7: JSON-encoded Integrity-Protected Incremental Tracing Profile

Appendix D. Example of the Integrity-Protected Pre-allocated Tracing Profile

A JSON [RFC7951] example of the Integrity-Protected Pre-allocated Tracing Profile is depicted in Figure 8. This configuration is received by an IOAM ingress node. This node first identifies the target flow by using the ACL parameter "test-acl" and then encapsulates the IOAM data in the NSH. The Integrity Protection Method to be used is method 0. The trace type indicates that each on-path node needs to capture the namespace-specific data in short format and add the data to the IOAM node data list. This node pre-allocates the node data list in the packet with 512 bytes.

```
{
  "ietf-ioam:ioam": {
    "admin-config": {
      "enabled": true
    },
    "profiles": {
      "profile": [
        {
          "profile-name": "ietf-test-profile",
          "filter": {
            "filter-type": "ietf-ioam:acl-filter",
            "ace-name": "test-acl"
          },
          "protocol-type": "ietf-ioam:nsh",
          "ietf-ioam-integrity:int-preallocated-tracing-profile": {
            "node-action": "ietf-ioam:action-encapsulate",
            "trace-types": {
              "use-namespace": "ietf-ioam:default-namespace",
              "trace-type": [
                "ietf-ioam:trace-namespace-data"
              ]
            },
            "max-length": 512,
            "int-method": "method-0"
          }
        }
      ]
    }
  }
}
```

Figure 8: JSON-encoded Integrity-Protected Pre-allocated Tracing Profile

Appendix E. Example of the Integrity-Protected Proof of Transit and Integrity-Protected Edge-to-Edge Profiles

A JSON [RFC7951] example of the Integrity-Protected Proof of Transit Profile, combined with the Integrity-Protected Edge-to-Edge Profile, is depicted in Figure 9. This configuration is received by an IOAM ingress node. This node encapsulates the Integrity-Protected Proof of Transit Type 0 in an IPv6 Hop-by-Hop Options header, and also encapsulates the Integrity-Protected Edge-to-Edge in an IPv6 Destination Options header. The Edge-to-Edge type indicates the presence of a 64-bit sequence number. The Integrity Protection Method to be used for both is method 0.

```
{
  "ietf-ioam:ioam": {
    "admin-config": {
      "enabled": true
    },
    "profiles": {
      "profile": [
        {
          "profile-name": "ietf-test-profile-pot",
          "protocol-type": "ietf-ioam:ipv6",
          "ietf-ioam-integrity:int-pot-profile": {
            "pot-type": "ietf-ioam:pot-type-0",
            "node-action": "ietf-ioam:action-encapsulate",
            "int-method": "method-0"
          }
        },
        {
          "profile-name": "ietf-test-profile-e2e",
          "protocol-type": "ietf-ioam:ipv6",
          "ietf-ioam-integrity:int-e2e-profile": {
            "node-action": "ietf-ioam:action-encapsulate",
            "e2e-types": {
              "use-namespace": "ietf-ioam:default-namespace",
              "e2e-type": [
                "ietf-ioam:e2e-seq-num-64"
              ]
            },
            "int-method": "method-0"
          }
        }
      ]
    }
  }
}
```

Figure 9: JSON-encoded Integrity-Protected Proof of Transit and Integrity-Protected Edge-to-Edge Profiles

Authors' Addresses

Justin Iurman

University of Liege

10, Allee de la decouverte (B28)

4000 Sart-Tilman

Belgium

Email: justin.iurman@uliege.be

Tianran Zhou

Huawei

156 Beiqing Rd.

Beijing

100095

China

Email: zhoutianran@huawei.com

APPENDIX C

This appendix contains a patch for the Linux kernel to add support for multi-queue (MQ) network devices in IOAM (*i.e.*, the queue depth data field), which is discussed in Chap. 9. The patch is based on *net-next v6.18-rc6* and, at the time of writing, has not been submitted yet. Performance evaluation is also included.

```
diff --git a/Documentation/networking/ioam6-sysctl.rst b/Documentation/
networking/ioam6-sysctl.rst
index c18cab2c481a..74495e324b39 100644
--- a/Documentation/networking/ioam6-sysctl.rst
+++ b/Documentation/networking/ioam6-sysctl.rst
@@ -24,3 +24,11 @@ ioam6_id_wide - INTEGER
     Define the wide IOAM id of this interface.

     Default is ~0.
+
+
+ioam6_aggr_tx_qdepth - BOOL
+
+    The value of the Queue Depth data field is an aggregation of all TX
+    queue depths (= enabled) or the depth of the selected TX queue only
+    (= disabled), specifically for this (egress) interface.
+
+    * 0 - disabled
+    * 1 - enabled (default)
diff --git a/include/linux/ipv6.h b/include/linux/ipv6.h
index 7294e4e89b79..be1b4c757e1b 100644
--- a/include/linux/ipv6.h
+++ b/include/linux/ipv6.h
@@ -88,6 +88,7 @@ struct ipv6_devconf {
     __u32          ioam6_id;
     __u32          ioam6_id_wide;
     __u8           ioam6_enabled;
+    __u8           ioam6_aggr_tx_qdepth;
```

```

        __u8          ndisc_evict_nocarrier;
        __u8          ra_honor_pio_life;
        __u8          ra_honor_pio_pflag;
diff --git a/include/uapi/linux/ipv6.h b/include/uapi/linux/ipv6.h
index d4d3ae774b26..5a35d2337456 100644
--- a/include/uapi/linux/ipv6.h
+++ b/include/uapi/linux/ipv6.h
@@ -200,6 +200,7 @@ enum {
        DEVCONF_ACCEPT_UNTRACKED_NA,
        DEVCONF_ACCEPT_RA_MIN_LFT,
        DEVCONF_FORCE_FORWARDING,
+       DEVCONF_IOAM6_AGGR_TX_QDEPTH,
        DEVCONF_MAX
};

diff --git a/net/ipv6/addrconf.c b/net/ipv6/addrconf.c
index b66217d1b2f8..ed1f1fd609c9 100644
--- a/net/ipv6/addrconf.c
+++ b/net/ipv6/addrconf.c
@@ -236,6 +236,7 @@ static struct ipv6_devconf ipv6_devconf __read_mostly = {
        .ioam6_enabled          = 0,
        .ioam6_id               = IOAM6_DEFAULT_IF_ID,
        .ioam6_id_wide          = IOAM6_DEFAULT_IF_ID_WIDE,
+       .ioam6_aggr_tx_qdepth   = 1,
        .ndisc_evict_nocarrier   = 1,
        .ra_honor_pio_life       = 0,
        .ra_honor_pio_pflag      = 0,
@@ -301,6 +302,7 @@ static struct ipv6_devconf ipv6_devconf_dflt __read_mostly
= {
        .ioam6_enabled          = 0,
        .ioam6_id               = IOAM6_DEFAULT_IF_ID,
        .ioam6_id_wide          = IOAM6_DEFAULT_IF_ID_WIDE,
+       .ioam6_aggr_tx_qdepth   = 1,
        .ndisc_evict_nocarrier   = 1,
        .ra_honor_pio_life       = 0,
        .ra_honor_pio_pflag      = 0,
@@ -5710,6 +5712,8 @@ static void ipv6_store_devconf(const struct ipv6_devconf
*cnf,

```

```

array[DEVCONF_IOAM6_ENABLED] = READ_ONCE(cnf->ioam6_enabled);
array[DEVCONF_IOAM6_ID] = READ_ONCE(cnf->ioam6_id);
array[DEVCONF_IOAM6_ID_WIDE] = READ_ONCE(cnf->ioam6_id_wide);
+ array[DEVCONF_IOAM6_AGGR_TX_QDEPTH] =
+     READ_ONCE(cnf->ioam6_aggr_tx_qdepth);
array[DEVCONF_NDISC_EVICT_NOCARRIER] =
    READ_ONCE(cnf->ndisc_evict_nocarrier);
array[DEVCONF_ACCEPT_UNTRACKED_NA] =
@@ -7267,6 +7271,15 @@ static const struct ctl_table addrconf_sysctl[] = {
        .mode                = 0644,
        .proc_handler         = proc_douintvec,
    },
+    {
+        .procname             = "ioam6_aggr_tx_qdepth",
+        .data                  = &ipv6_devconf.ioam6_aggr_tx_qdepth,
+        .maxlen                = sizeof(u8),
+        .mode                  = 0644,
+        .proc_handler          = proc_dou8vec_minmax,
+        .extra1                 = (void *)SYSCTL_ZERO,
+        .extra2                 = (void *)SYSCTL_ONE,
+    },
+    {
        .procname              = "ndisc_evict_nocarrier",
        .data                   = &ipv6_devconf.ndisc_evict_nocarrier,
diff --git a/net/ipv6/ioam6.c b/net/ipv6/ioam6.c
index 9553a3200081..806613915a98 100644
--- a/net/ipv6/ioam6.c
+++ b/net/ipv6/ioam6.c
@@ -690,6 +690,24 @@ struct ioam6_namespace *ioam6_namespace(struct net *net,
    __be16 id)
    {
        return rhashtable_lookup_fast(&nsdata->namespaces, &id, rht_ns_params);
    }

+static void __ioam6_sum_qdepth(struct net_device *dev,
+    struct netdev_queue *txq,
+    void *arg)
+{
+    struct Qdisc *qdisc;

```

```

+     u32 qlen, backlog;
+     u64 *sum;
+
+     qdisc = rcu_dereference(txq->qdisc);
+
+     spin_lock_bh(qdisc_lock(qdisc));
+     qdisc_qstats_qlen_backlog(qdisc, &qlen, &backlog);
+     spin_unlock_bh(qdisc_lock(qdisc));
+
+     sum = (u64 *)arg;
+     *sum += backlog;
+}
+
static void __ioam6_fill_trace_data(struct sk_buff *skb,
                                   struct ioam6_namespace *ns,
                                   struct ioam6_trace_hdr *trace,
@@ -780,19 +798,33 @@ static void __ioam6_fill_trace_data(struct sk_buff *skb,

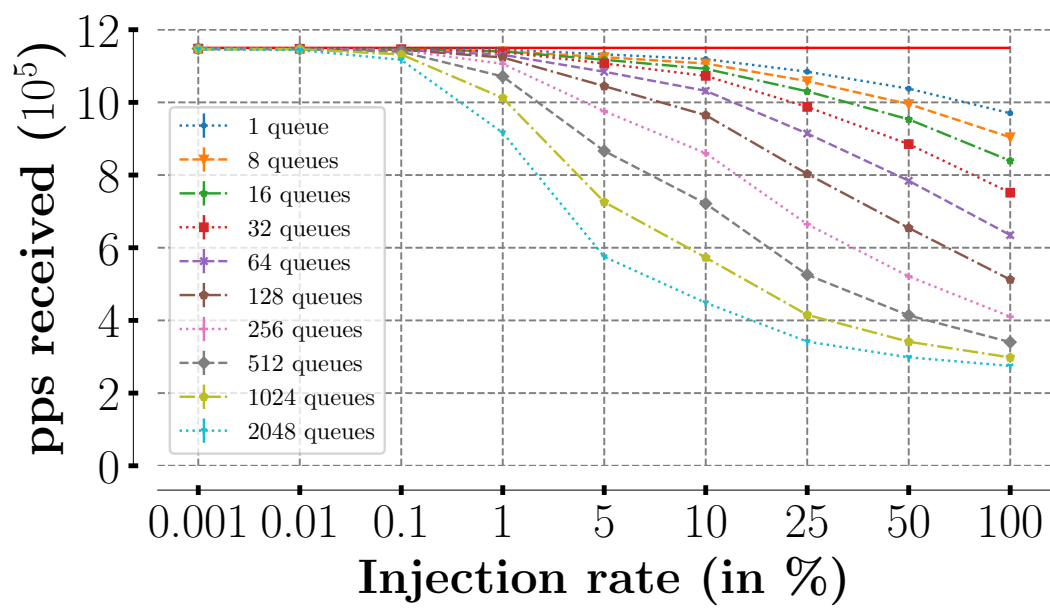
/* queue depth */
if (trace->type.bit6) {
-     struct netdev_queue *queue;
-     struct Qdisc *qdisc;
-     __u32 qlen, backlog;
-
-     if (dev->flags & IFF_LOOPBACK) {
-         *(__be32 *)data = cpu_to_be32(IOAM6_U32_UNAVAILABLE);
+     raw32 = IOAM6_U32_UNAVAILABLE;
+     } else {
-         queue = skb_get_tx_queue(dev, skb);
-         qdisc = rcu_dereference(queue->qdisc);
-         qdisc_qstats_qlen_backlog(qdisc, &qlen, &backlog);
+         raw64 = 0;
+
+         if (READ_ONCE(__in6_dev_get(dev)->cnf.
ioam6_aggr_tx_qdepth)) {
+             netdev_for_each_tx_queue(dev,
+                                     __ioam6_sum_qdepth,
+                                     &raw64);

```

```

-             *(__be32 *)data = cpu_to_be32(backlog);
+             raw32 = (__force u32)(raw64 / dev->
num_tx_queues);
+         } else {
+             __ioam6_sum_qdepth(dev,
+                                 skb_get_tx_queue(dev, skb),
+                                 &raw64);
+
+             raw32 = (__force u32)raw64;
+         }
+
+         /* Since IOAM6_U32_UNAVAILABLE == U32_MAX (as per
+          * RFC9197), we need to handle this (unlikely) case.
+          */
+         if (unlikely(raw32 == U32_MAX))
+             raw32--;
+     }
+
+     *(__be32 *)data = cpu_to_be32(raw32);
+     data += sizeof(__be32);
+ }

```



APPENDIX D

This appendix contains the latest version [34] of the entire IETF draft about integrity protection of IOAM (version 19, at the time of writing), which includes all the recommendations and improvements suggested in Chap. 11. This IETF work has just been approved and is about to become an IETF RFC .

Workgroup: ippm
Internet-Draft: draft-ietf-ippm-ioam-data-integrity-19
Published: 11 May 2026
Intended Status: Standards Track
Expires: 12 November 2026
Authors: F. Brockners S. Bhandari T. Mizrahi J. Iurman
 Cisco Databricks Huawei University of Liege

Integrity Protection of In Situ Operations, Administration, and Maintenance (IOAM) Data Fields

Abstract

In Situ Operations, Administration, and Maintenance (IOAM) records operational (including telemetry) information in packets while they traverse a path in the network. RFC 9197 specifies data fields for IOAM (a.k.a IOAM-Data-Fields) and associated data types. This document specifies integrity protection of IOAM-Data-Fields for Intra-IOAM-Domain use cases.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 12 November 2026.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction
2. Conventions
 - 2.1. Requirements Language
 - 2.2. Terminology
3. Threat Analysis
 - 3.1. Modification: IOAM-Data-Fields
 - 3.2. Modification: IOAM Option-Type header
 - 3.3. Injection: IOAM-Data-Fields
 - 3.4. Injection: IOAM Option-Type header
 - 3.5. Deletion: IOAM-Data-Fields
 - 3.6. Deletion: IOAM Option-Type header
 - 3.7. Replay
 - 3.8. Management and Exporting
 - 3.9. Delay
 - 3.10. Threat Summary
4. Integrity-Protected Option-Types
 - 4.1. Integrity-Protected Trace Option-Types
 - 4.2. Integrity-Protected POT Option-Type
 - 4.3. Integrity-Protected E2E Option-Type
5. Integrity Protection Method
 - 5.1. Key and Nonce Management
 - 5.2. Integrity Protection of Header Fields
 - 5.3. Encapsulating Node
 - 5.4. Transit Node
 - 5.5. Decapsulating Node
 - 5.6. Validator
6. IANA Considerations
 - 6.1. IOAM Option-Types

6.2. IOAM Integrity Protection Methods

7. Operational Considerations

7.1. Manageability

7.2. Scalability and Performance

7.3. Migration

7.4. MTU

8. Security Considerations

9. Acknowledgements

10. References

10.1. Normative References

10.2. Informative References

Authors' Addresses

1. Introduction

In Situ Operations, Administration, and Maintenance (IOAM) [RFC9197] records OAM information within packets while they traverse a network domain. The term "In Situ" refers to the fact that the OAM data is added to the data packets rather than being sent within packets specifically dedicated to OAM (a.k.a. In-Data-Packet OAM [I-D.ietf-opsawg-oam-characterization]).

IOAM is deployed inside an IOAM-Domain, which consists of a set of nodes that use IOAM. The boundaries of the IOAM-Domain are formed by IOAM control points, as defined in [RFC9197]. It is expected that all nodes in an IOAM-Domain are managed by the same administrative entity, that has means to select, monitor, and control access to all the networking devices. Per [RFC9197], IOAM-Data-Fields are carried in the clear within packets and there are no protections against any device tampering with the data. IOAM-Data-Fields collected in an untrusted environment require at least integrity protection to support critical operational decisions. Refer to [RFC9197] for more details on IOAM-Domains.

Since arbitrary nodes can tamper with all packets data, including IOAM-Data-Fields, and the packets are (in general) processed by other intermediary nodes before they are delivered to a node that can verify the IOAM fields of the packet, there is little value in attempting to use cryptographic mechanisms to prevent such modifications to the IOAM fields in the packet. Instead, this document is limited to the "detectability problem", namely, to allow an endpoint to

detect that such modification has occurred since the generation of the IOAM-Data-Fields. In addition, the following considerations and requirements are to be taken into account in constructing an IOAM integrity mechanism:

1. IOAM data is processed by the data plane, hence viability of any method to prove integrity of the IOAM-Data-Fields must be feasible at data plane processing/forwarding rates (IOAM may be applied to all traffic that a router forwards).
2. IOAM data is carried within packets. The additional space required to provide integrity protection of the IOAM-Data-Fields must not cause packets to exceed the applicable Maximum Transmission Unit (MTU) within the IOAM-Domain and should minimize adverse effects on packet processing.
3. Protection against replay of old IOAM data should be possible. Without replay protection, a rogue node can present the old IOAM data, masking any ongoing network issues/activity and making the IOAM-Data-Fields collection useless.

This document defines a method to protect the integrity of IOAM-Data-Fields for Intra-IOAM-Domain use cases, using the IOAM Option-Types specified in [\[RFC9197\]](#) as an example.

2. Conventions

2.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [\[RFC2119\]](#) [\[RFC8174\]](#) when, and only when, they appear in all capitals, as shown here.

2.2. Terminology

This document uses terms such as IOAM-Data-Fields, IOAM-Domain, IOAM-Namespace, IOAM-Option-Type (or Option-Type), encapsulating node, transit node, and decapsulating node. Refer to [\[RFC9197\]](#) for definitions.

This document also introduces the term "Validator". Refer to [Section 5.6](#) for the definition.

The following abbreviations are used in this document:

OAM:	Operations, Administration, and Maintenance
IOAM:	In Situ OAM
POT:	Proof of Transit
E2E:	Edge to Edge
MTU:	Maximum Transmission Unit
GCM:	Galois/Counter Mode

GMAC: Galois Message Authentication Code
IV: Initialization Vector
ICV: Integrity Check Value
AAD: Additional Authenticated Data

The following notation is used in this document:

A || B The concatenation of A and B, where the octets of B immediately follow the octets of A.

3. Threat Analysis

This section presents a threat analysis of integrity-related threats in the context of IOAM. The threats that are discussed are assumed to be independent of the lower layer protocols; it is assumed that threats at other layers are handled by security mechanisms that are deployed at those layers.

This document is focused on integrity protection for IOAM-Data-Fields. Thus, the threat analysis includes threats that are related to or result from compromising the integrity of IOAM-Data-Fields. Other security aspects such as confidentiality are out of scope.

Throughout the analysis there is a distinction between on-path and off-path attackers. As discussed in [RFC9055], on-path attackers are located in a position that allows interception, modification, or dropping of in-flight protocol packets, whereas off-path attackers can only attack by generating protocol packets. Since IOAM operates within administratively controlled IOAM-Domains that define a trust boundary, this reduces potential attack vectors and should naturally mitigate off-path threats.

The analysis also includes the impact of each of the threats. Generally speaking, the impact of a successful attack on an OAM protocol [RFC7276] is an illusion of nonexistent failures (or disruption) or preventing the detection of actual ones, as described in Section 9 of [RFC9197]; in both cases, the attack could result in Denial-of-Service (DoS). Furthermore, creating the illusion of a nonexistent issue could trigger unnecessary processing in some of the IOAM nodes along a forwarding path, and could cause more IOAM-related data to be exported to the management plane than is conventionally necessary. Beyond these general impacts, threat-specific impacts are discussed in each of the subsections below.

3.1. Modification: IOAM-Data-Fields

Applicability

On-path only.

Threat

An attacker can modify the IOAM-Data-Fields of in-transit packets. The modification can either be applied to all packets or selectively applied to a subset. Maliciously modified IOAM-Data-Fields can, for example, mislead network diagnostics, result in incorrect network performance metrics, or could misguide network optimization efforts.

Impact

By systematically modifying the IOAM-Data-Fields of some or all of the in-transit packets, an attacker can create a fake picture of the network status. Potential consequences include an impact on network performance, a change in the recorded forwarding path of packets, either based on fake node positions or fake data provided by the attacker to fool the system that ingests IOAM-Data-Fields.

3.2. Modification: IOAM Option-Type header

Applicability

On-path only.

Threat

An attacker can modify the fields of an IOAM Option-Type header to change or disrupt the behavior of nodes processing IOAM-Data-Fields along the path, or change the interpretation of IOAM-Data-Fields. The modification can either be applied to all packets or selectively applied to a subset.

Impact

Modification of fields in an IOAM Option-Type header can have multiple impacts. The following examples are non-exhaustive.

An attacker that modifies the Namespace-ID field ([RFC9197], Section 4.3), common to all IOAM Option-Type headers, can alter the definition and interpretation of IOAM-Data-Fields. Without the correct Namespace-ID, IOAM-Data-Fields cannot be reliably interpreted.

An attacker that modifies the Trace-Type field ([RFC9197], Section 4.4.1) of the IOAM Trace Option-Type header can increase node processing overhead for IOAM-Data-Fields and increase on-the-wire overhead.

An attacker that modifies the RemainingLen field ([RFC9197], Section 4.4.1) of the IOAM Trace Option-Type header can prevent nodes from incorporating IOAM-Data-Fields.

An attacker that sets the Loopback flag ([RFC9322], Section 3) in the IOAM Trace Option-Type header can cause a Denial-of-Service by inducing each node to send packet copies to the encapsulating node.

Modification of the header can result in impacts similar to those described in [Section 3.1](#).

3.3. Injection: IOAM-Data-Fields

Applicability

On-path only.

Threat

An attacker can inject additional IOAM-Data-Fields into packets containing at least one IOAM Option-Type, thus falsifying the view of the actual network state. The injection can either be applied to all packets or selectively applied to a subset.

Impact

This attack causes impacts similar to those described in [Section 3.1](#).

3.4. Injection: IOAM Option-Type header

Applicability

Both on-path and off-path.

Threat

An attacker can inject packets with IOAM Option-Type headers, thus manipulating other nodes that process IOAM-Data-Fields in the network.

Impact

This attack and its impacts are similar to those described in [Section 3.2](#).

3.5. Deletion: IOAM-Data-Fields

Applicability

On-path only.

Threat

An attacker can remove IOAM-Data-Fields from packets containing at least one IOAM Option-Type, thus hiding the diagnosis of some nodes. The deletion can either be applied to all packets or selectively applied to a subset.

Impact

This attack causes impacts similar to those described in [Section 3.1](#).

3.6. Deletion: IOAM Option-Type header

Applicability

On-path only.

Threat

An attacker can remove IOAM Option-Type headers from packets, thus preventing the use of IOAM to diagnose the network. The deletion can either be applied to all packets or selectively applied to a subset. The mechanisms in this document do not provide any mitigation against this threat.

Impact

By systematically removing IOAM Option-Type headers from some or all of the in-transit packets, an attacker can make telemetry recording incomplete or even impossible. As a consequence, network diagnosis could be incomplete or non-existent.

3.7. Replay

Applicability

Both on-path and off-path.

Threat

In addition to replaying old packets in general, an attacker can replay packets with IOAM-Data-Fields. Specifically, an attacker could replay a previously transmitted IOAM Option-Type header with a new data packet, therefore attaching old IOAM-Data-Fields to a fresh user packet.

Impact

This attack causes impacts similar to those described in [Section 3.1](#).

3.8. Management and Exporting

Applicability

Both on-path and off-path.

Threat

Attacks that compromise the integrity of IOAM-Data-Fields can be applied at the management plane, e.g., by manipulating network management packets. Furthermore, the integrity of IOAM-Data-Fields that are exported to a receiving entity can also be compromised. Management plane attacks are out of scope; the network management protocol is expected to include inherent security capabilities. The integrity of exported data

is also out of scope. It is expected that the specification of the export format will discuss the relevant security aspects.

Impact

Malicious manipulation of the management protocol can cause nodes that process IOAM-Data-Fields to malfunction, to be overloaded, or to incorporate unnecessary IOAM-Data-Fields into user packets. The impact of compromising the integrity of exported IOAM-Data-Fields is similar to the impacts of previous threats that were described in this section.

3.9. Delay

Applicability

On-path only.

Threat

An attacker might delay some or all of the in-transit packets that include IOAM-Data-Fields to create an illusion of congestion. Delay attacks are well known in the context of deterministic networks [[RFC9055](#)] and time synchronization [[RFC7384](#)], and could be somewhat mitigated in these environments by using redundant paths in a way that is resilient to an attack along one of the paths. This approach does not address the threat in the context of IOAM, as it does not meet the requirement to measure a specific path or to detect a problem along the path. Note that the mechanisms in this document do not attempt to provide any mitigation against this threat.

Impact

Since IOAM can be applied to a fraction of the traffic, an attacker can detect and delay only the packets that include IOAM-Data-Fields, thus preventing the authenticity of delay and load measurements.

3.10. Threat Summary

Threat	Document Scope	
	In	Out
Modification: IOAM-Data-Fields	X	
Modification: IOAM Option-Type header	X	
Injection: IOAM-Data-Fields	X	
Injection: IOAM Option-Type header	X	
Deletion: IOAM-Data-Fields	X	
Deletion: IOAM Option-Type header		X
Replay	X	
Management and Exporting		X
Delay		X

Figure 1: Threat Analysis Summary

4. Integrity-Protected Option-Types

This section defines new IOAM Option-Types to carry IOAM-Data-Fields with integrity protection. For each IOAM Option-Type defined in [RFC9197], a corresponding Integrity-Protected Option-Type is defined as follows:

- IOAM Integrity-Protected Pre-allocated Trace Option-Type: corresponds to the IOAM Pre-allocated Trace Option-Type ([RFC9197], Section 4.4) with integrity protection.
- IOAM Integrity-Protected Incremental Trace Option-Type: corresponds to the IOAM Incremental Trace Option-Type ([RFC9197], Section 4.4) with integrity protection.
- IOAM Integrity-Protected POT Option-Type: corresponds to the IOAM POT Option-Type ([RFC9197], Section 4.5) with integrity protection.
- IOAM Integrity-Protected E2E Option-Type: corresponds to the IOAM E2E Option-Type ([RFC9197], Section 4.6) with integrity protection.

The Direct Export (DEX) Option-Type [RFC9326] is not covered by the Integrity Protection Method defined in Section 5. This document focuses on the integrity protection of IOAM-Data-Fields, whereas DEX does not carry IOAM-Data-Fields by definition. As a consequence, DEX and similar (i.e., any IOAM Option-Type with no IOAM-Data-Fields) are considered out of scope and MUST NOT use the Integrity Protection Method defined in this document.

The Integrity Protection header sits between an IOAM Option-Type header and its IOAM-Data-Fields, forming an equivalent Integrity-Protected Option-Type. This placement ensures that the Integrity Protection header is part of the header, rather than a trailer after the IOAM-Data-Fields. The Integrity Protection header is defined as shown in Figure 2.

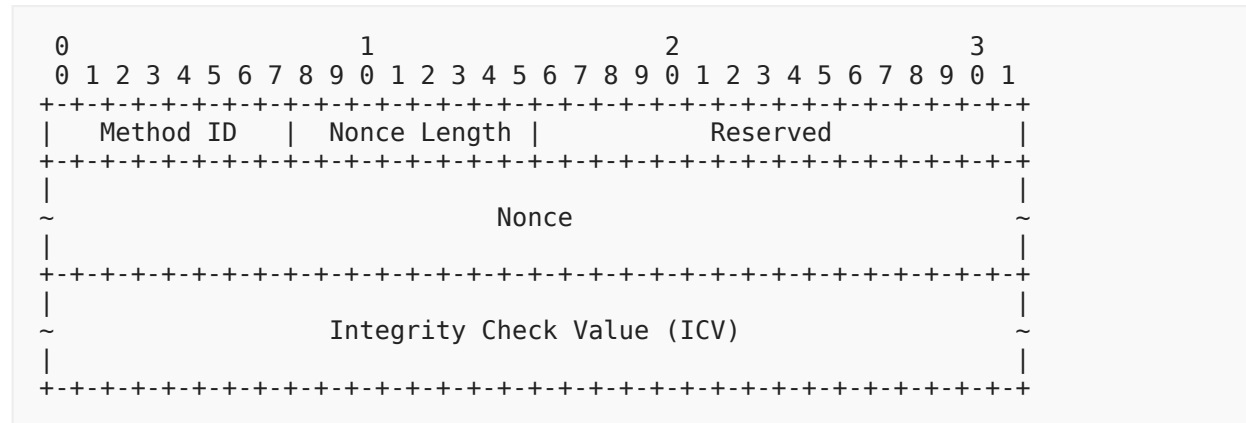


Figure 2: Integrity Protection header

Method ID: 8-bit unsigned integer, see Section 6.2. It defines the Integrity Protection Method to compute the Integrity Check Value (ICV) field. If a node encounters an unknown value, it MUST NOT change the contents of the Integrity Protection header and MUST NOT change the contents of the IOAM-Data-Fields. In other words, the node must not process the IOAM Option-Type.

Nonce Length: 8-bit unsigned integer. It defines the length of the Nonce field, in octets.

Reserved: 16-bit Reserved field. It MUST be set to zero upon transmission and ignored upon receipt.

Nonce: Variable length field. Its size depends on the Nonce Length field.

Integrity Check Value (ICV): Variable length field. Its size depends on the Method ID field.

In order to keep IOAM-Data-Fields aligned ([RFC9197], Section 4.4.2), the total length of the Integrity Protection header MUST be a multiple of 4 octets.

4.1. Integrity-Protected Trace Option-Types

Both the IOAM Pre-allocated Trace Option-Type header and the IOAM Incremental Trace Option-Type header are identical, as defined in Section 4.4 of [RFC9197]. When followed by the Integrity Protection header, they respectively form the IOAM Integrity-Protected Pre-allocated Trace Option-Type and the IOAM Integrity-Protected Incremental Trace Option-Type (Figure 3). The definitions of fields that are not part of the Integrity Protection header are the same as in [RFC9197].

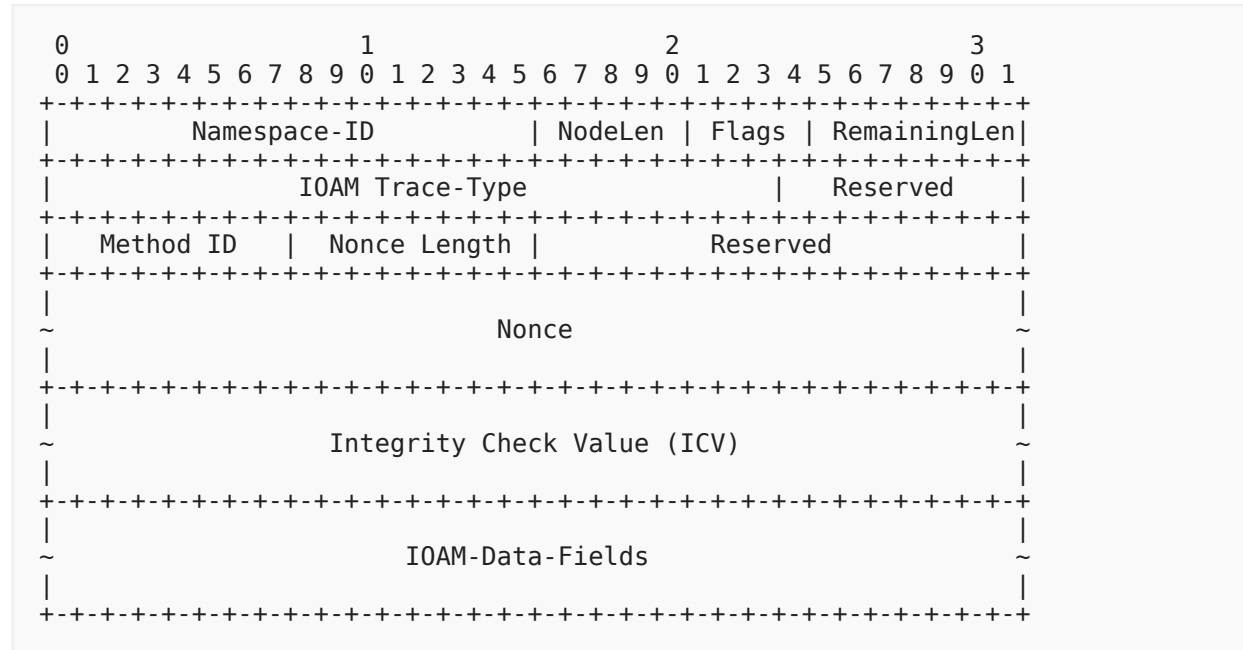


Figure 3: Integrity-Protected Trace Option-Types header

4.2. Integrity-Protected POT Option-Type

The IOAM POT Option-Type header is defined in Section 4.5 of [RFC9197]. When followed by the Integrity Protection header, it forms the IOAM Integrity-Protected POT Option-Type (Figure 4). The definitions of fields that are not part of the Integrity Protection header are the same as in [RFC9197].

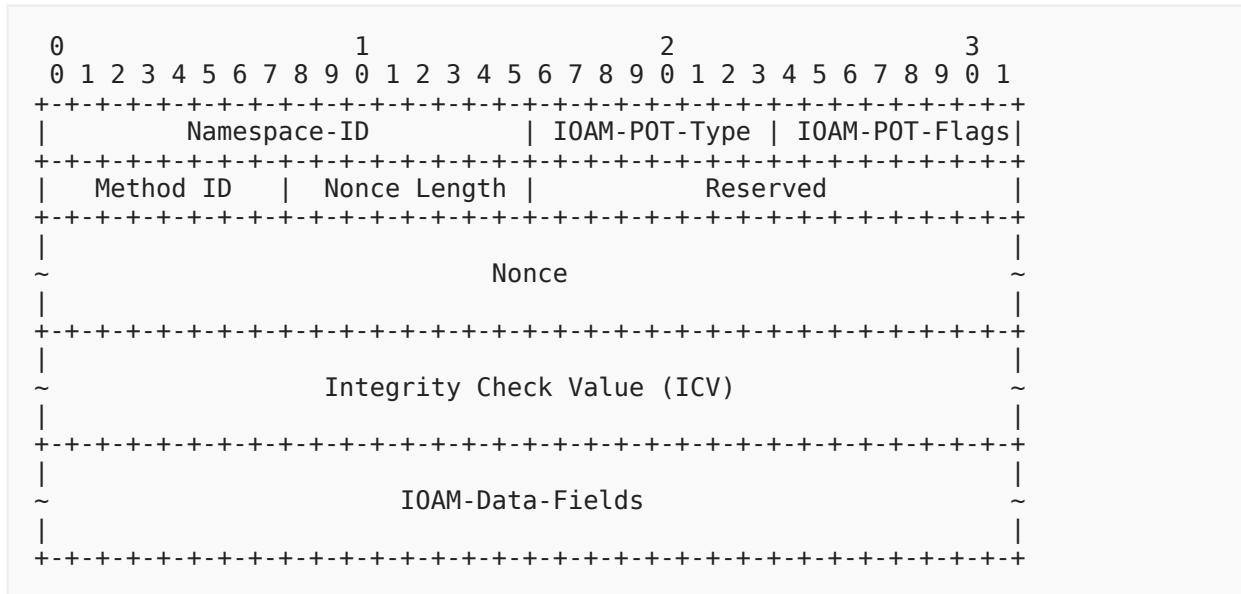


Figure 4: Integrity-Protected POT Option-Type header

4.3. Integrity-Protected E2E Option-Type

The IOAM E2E Option-Type header is defined in Section 4.6 of [RFC9197]. When followed by the Integrity Protection header, it forms the IOAM Integrity-Protected E2E Option-Type (Figure 5). The definitions of fields that are not part of the Integrity Protection header are the same as in [RFC9197].

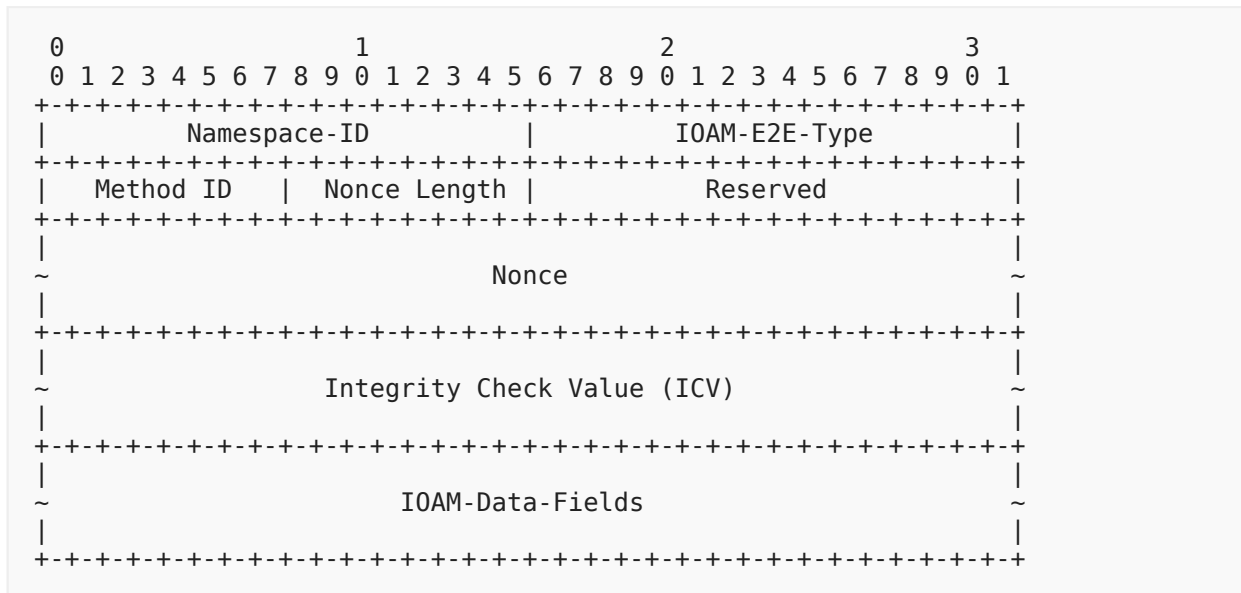


Figure 5: Integrity-Protected E2E Option-Type header

5. Integrity Protection Method

This document defines an Integrity Protection Method for IOAM-Data-Fields using symmetric keys.

The method uses AES-GMAC [AES] [NIST.800-38D], a mode of operation that provides data origin authentication and is a specialization of Galois/Counter Mode (GCM). GCM takes as input a secret key, an Initialization Vector (IV), a plaintext, and Additional Authenticated Data (AAD), and produces a ciphertext and an Authentication Tag. GMAC is the special case of GCM with zero-length plaintext; the ciphertext output is empty and ignored, and only the Authentication Tag is produced.

For this method, the Authentication Tag **MUST NOT** be truncated and **MUST** be 16 octets in length.

In this document, the GMAC Initialization Vector (IV) is referred to as the nonce, the GMAC Authentication Tag is referred to as the Integrity Check Value (ICV), and the GMAC Additional Authenticated Data (AAD) is referred to as AAD.

Due to IOAM space constraints, maintaining a separate nonce and ICV for each IOAM node participating in integrity protection is not practical. This Integrity Protection Method uses a single Nonce field and a single ICV field in the packet. The nonce is generated by the encapsulating node according to the requirements defined in [Section 5.1](#) and remains unchanged. The ICV is iteratively updated by participating nodes using the received ICV and their immutable IOAM-Data-Fields as inputs to the GMAC operation. This design enables reconstruction of the integrity chain by a Validator.

5.1. Key and Nonce Management

In order to use this method and apply integrity protection, it is **REQUIRED** that each IOAM node that updates the ICV (in the Integrity Protection header) has its own unique symmetric key. Although GMAC supports all AES key sizes (i.e., 128, 192, and 256 bits), it is **RECOMMENDED** to use the longest key size when possible. Each key **MUST** be securely generated and fresh. Also, each key **MUST** be securely distributed to only the corresponding IOAM nodes and any Validator that needs to validate messages protected by that key. Except key rotation requirements, the details of key generation and distribution are out of scope.

In addition to key management, per-message nonces used with GMAC **MUST** be managed to prevent reuse of a key-nonce pair. Since reuse of a nonce with a given key allows forgery of arbitrary ciphertexts with valid authentication tag, it is extremely important to have high confidence in nonce non-reuse.

For this method, the size of the nonce **MUST** always be 12 octets. If a node receives a Nonce Length value other than 12, it **MUST NOT** change the contents of the Integrity Protection header and **MUST NOT** change the contents of the IOAM-Data-Fields. In other words, the node must not process the IOAM Option-Type. A nonce **MUST NOT** be reused with the same key. The nonce is based on the "Deterministic Construction" [NIST.800-38D] and has the format shown in [Figure 6](#).

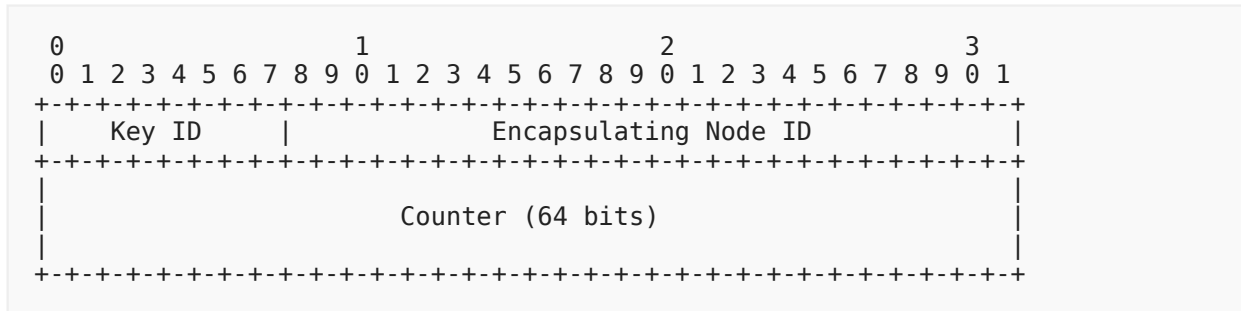


Figure 6: Structure of the Nonce field (12 octets)

Key ID: 8-bit unsigned integer. It identifies the key used by the corresponding Encapsulating Node ID to compute the ICV with GMAC.

Encapsulating Node ID: 24-bit unsigned integer. It identifies an IOAM encapsulating node that generated the nonce. It **MUST** be unique for each IOAM node in the IOAM-Domain, which means a reasonable limit of 2^{24} distinct IOAM nodes in total.

Counter: 64-bit unsigned integer. It is an incrementing counter managed by the corresponding encapsulating node (i.e., the one whose ID is the same as the Encapsulating Node ID field). The counter **MUST** start at 0 for each Key ID and **MUST** be unique for each packet, which means a limit of 2^{64} packets per Key ID on the corresponding Encapsulating Node ID. Since GMAC does not require a nonce to be secret or unpredictable, it is secure to use a counter.

When the key-specific counter of an encapsulating node reaches the maximum value (i.e., 2^{64}), the encapsulating node **MUST** stop applying integrity protection with that key and start using a new one. Ideally, before that limit is reached, the key management system **SHOULD** rotate the key and notify any Validator that needs to validate messages protected by that key. The Key ID field provides an easy way to avoid conflicts during key rotations with other IOAM nodes that apply integrity protection. When an encapsulating node reaches its maximum use of 256 distinct keys (i.e., see the Key ID field) and is also about to reach the limit of its key-specific counter, the key management system **MUST** rotate the keys of all IOAM nodes in the IOAM-Domain and notify any Validator with the new keys. In addition, the internal integrity protection state of all IOAM nodes, which is useful to detect and prevent key-nonce reuse, **MUST** be reset. Overall, the key management system **MUST NOT** (ever) redistribute an old key to any of the IOAM nodes in the IOAM-Domain.

An encapsulating node that participates in the integrity protection of IOAM-Data-Fields **MAY** retain on persistent storage the current key in use and the corresponding key-specific counter, such that a power interruption or system crash (or reboot in general) does not lead to nonce reuse. If it is not possible for some reason, the encapsulating node **MUST NOT** reuse the key and the key management system **MUST** rotate the key before the encapsulating node resumes integrity protection. In case of a power interruption or system crash (or reboot in general) on any transit node that participates in the integrity protection of IOAM-Data-Fields, the transit node **MUST NOT** reuse the key and the key management system **MUST** rotate the key before the transit node resumes integrity protection.

To illustrate with an example, it would take 7 years for a single key of an encapsulating node to reach the limit of 2^{64} 1500-byte packets on a 1 Pbps (Petabits per second) link at line rate, while it would take approximately 143 days with 64-byte packets. For 256 distinct keys of an encapsulating node, it would take 1792 and 100 years respectively. This is a worst-case scenario, as such link capacity is not common and IOAM might not be applied to all packets.

5.2. Integrity Protection of Header Fields

The main objective of the Integrity Protection Method defined in this document is to provide integrity protection of IOAM-Data-Fields. However, some Option-Type header fields are crucial for IOAM-Data-Fields (e.g., the IOAM Namespace-ID, which is common to all IOAM Option-Types). Without such fields, IOAM-Data-Fields cannot be reliably interpreted. As a consequence, the integrity of any immutable Option-Type header field **MUST** be protected. As for mutable Option-Type header fields, they do not benefit from any integrity protection since their values can change.

With GMAC, the bit length of the AAD **MUST** be a multiple of 8. In other words, the AAD is made up of whole octets. Masking specific bits of Option-Type header fields allows this constraint to be respected, even for fields that are single bits (e.g., flags) or not aligned on natural boundaries. The list of Option-Type header fields and their corresponding integrity protection masks to be applied (i.e., a logical AND operation) are defined in [Section 6.2](#).

For interoperability, having a dynamic registry to specify which header fields participate in integrity protection might not seem optimal at first glance. However, from a vendor perspective, integrity protection is simply a feature built on top of IOAM. Therefore, if two different vendors providing IOAM integrity protection are not interoperable, it probably means that their core IOAM implementations are not interoperable in the first place. Since IOAM has no versions, it does not make sense to introduce versions in the integrity protection feature either. This is the reason why it is acceptable to have a dynamic registry for this purpose and to update it accordingly over time.

5.3. Encapsulating Node

The encapsulating node **MUST** follow the instructions in [Section 5.1](#) to generate a new nonce, which is then stored in the Nonce field of the Integrity Protection header (the Nonce Length field is set accordingly). In particular, nonce generation **MUST** comply with the nonce uniqueness requirements defined in [Section 5.1](#). The Method ID field **MUST** be set to 0, as defined in [Section 6.2](#).

The initial ICV is the result of a GMAC operation, using the encapsulating node's key and the nonce generated by the encapsulating node. The AAD consists of a list of immutable header fields as defined in [Section 5.2](#) (depending on the Integrity-Protected Option-Type) and immutable IOAM-Data-Fields added by the encapsulating node. In the case where the Integrity-Protected

Pre-allocated Trace Option-Type is used, the encapsulating node includes the IOAM-Data-Fields that correspond to itself, i.e., "node data list [n]" ([RFC9197], Section 4.4). The encapsulating node performs the following operations:

- AAD = (Header Fields || IOAM-Data-Fields)
- ICV = GMAC(Key, Nonce, AAD)

Fields in the AAD are encoded in network byte order and in the same sequence as they appear on the wire. The encapsulating node then stores the ICV in the corresponding field of the Integrity Protection header.

5.4. Transit Node

A transit node MUST NOT generate a new nonce, as this would break the integrity chain and prevent validation of IOAM-Data-Fields contributed by previous nodes. It MUST use the received Nonce field for its own GMAC operation. A transit node MUST NOT reuse a nonce with the same key, as this is critical for GMAC security. Therefore, it MUST be able to detect reuse of nonces with its current key. Details on how this detection is implemented are out of scope. If a transit node receives a Nonce field that has already been used with its current key, it MUST NOT change the contents of the Integrity Protection header and MUST NOT change the contents of the IOAM-Data-Fields. In other words, the transit node must not process the IOAM Integrity-Protected Option-Type.

The ICV is the result of a GMAC operation, using the transit node's key and the received nonce. The AAD consists of the received ICV (ICV_rx) and immutable IOAM-Data-Fields added by the transit node. In the case where the Integrity-Protected Pre-allocated Trace Option-Type is used, the transit node includes the IOAM-Data-Fields that correspond to itself, i.e., "node data list [n-i]" ([RFC9197], Section 4.4). The transit node performs the following operations:

- AAD = (ICV_rx || IOAM-Data-Fields)
- ICV = GMAC(Key, Nonce, AAD)

Fields in the AAD are encoded in network byte order and in the same sequence as they appear on the wire. The transit node then updates the ICV field accordingly in the Integrity Protection header.

If the transit node does not add any immutable IOAM-Data-Fields (e.g., it only modifies mutable IOAM-Data-Fields or does nothing), and if the transit node, in case the Integrity-Protected Pre-allocated Trace Option-Type is used, does not update the "node data list" array, then the transit node MUST NOT update the ICV field in the Integrity Protection header.

A transit node MUST NOT add or remove the Integrity Protection header. Also, a transit node MUST NOT modify the Method ID field, the Nonce Length field, or the Nonce field in the Integrity Protection header.

A transit node does not validate the integrity of previously added IOAM-Data-Fields. Integrity validation is exclusively performed by a Validator as defined in [Section 5.6](#).

5.5. Decapsulating Node

The decapsulating node MAY perform the function of the Validator. If it does, ignore this section and refer directly to [Section 5.6](#).

If the decapsulating node does not perform the function of the Validator, which is an alternative to put any Validator out of the forwarding path in case of performance concerns, the decapsulating node MUST send the entire IOAM Integrity-Protected Option-Type to a Validator. The method to send it to a Validator is out of scope.

The decapsulating node MUST NOT generate a new nonce, as this would break the integrity chain and prevent validation of IOAM-Data-Fields contributed by previous nodes. It MUST use the received Nonce field for its own GMAC operation. A decapsulating node MUST NOT reuse a nonce with the same key, as this is critical for GMAC security. Therefore, it MUST be able to detect reuse of nonces with its current key. Details on how this detection is implemented are out of scope. If a decapsulating node receives a Nonce field that has already been used with its current key, it MUST NOT change the contents of the Integrity Protection header and MUST NOT change the contents of the IOAM-Data-Fields. In other words, the decapsulating node must not process the IOAM Integrity-Protected Option-Type.

The decapsulating node updates the ICV field in the Integrity Protection header. The ICV is the result of a GMAC operation, using the decapsulating node's key and the received nonce. The AAD consists of the received ICV (ICV_rx) and immutable IOAM-Data-Fields added by the decapsulating node. In the case where the Integrity-Protected Pre-allocated Trace Option-Type is used, the decapsulating node includes the IOAM-Data-Fields that correspond to itself, i.e., "node data list [n-i]" ([\[RFC9197\]](#), Section 4.4). The decapsulating node performs the following operations:

- AAD = (ICV_rx || IOAM-Data-Fields)
- ICV = GMAC(Key, Nonce, AAD)

Fields in the AAD are encoded in network byte order and in the same sequence as they appear on the wire.

If the decapsulating node does not add any immutable IOAM-Data-Fields (e.g., it only modifies mutable IOAM-Data-Fields or does nothing), and if the decapsulating node, in case the Integrity-Protected Pre-allocated Trace Option-Type is used, does not update the "node data list" array, then the decapsulating node MUST NOT update the ICV field in the Integrity Protection header.

The decapsulating node MUST NOT add the Integrity Protection header. Also, the decapsulating node MUST NOT modify the Method ID field, the Nonce Length field, or the Nonce field in the Integrity Protection header.

5.6. Validator

An IOAM node that performs the validation of the integrity protection is referred to as a Validator. Any Validator is a key trusted entity in this system, as it has access to all of the keying material in use and makes the final determination of whether an ICV is valid.

A validator is not vulnerable to key-nonce reuse, since the computed ICV remains internal. However, a protection against replay attacks in general (more specifically, replay of old IOAM-Data-Fields) is still needed. To this end, a Validator **MUST** be able to detect nonces already used with specific keys during validation to prevent replays. Details on how this detection is implemented are out of scope. If a Validator receives a Nonce field that has already been used with a specific key during validation, it **MUST** consider the ICV as invalid and ignore the next steps.

To validate an ICV, a Validator **MUST** recompute the cumulative ICV by replaying the sequence of GMAC operations in the same order as performed by the IOAM nodes, using the IOAM-Data-Fields contained in the packet and the corresponding keys associated with each node. Since only the latest ICV is carried, each step uses the ICV produced by the previous step as input to the next computation. The recomputed ICV is then compared with the ICV field (ICV_rx) carried in the packet. The Validator performs the following operations:

Step 0 (encapsulating node)

- AAD_0 = (Header Fields || IOAM-Data-Fields_0)
- ICV_0 = GMAC(Key_0, Nonce, AAD_0)

Step i (i-th node)

- AAD_i = (ICV_{i-1} || IOAM-Data-Fields_i)
- ICV_i = GMAC(Key_i, Nonce, AAD_i)

Step n (n-th node, i.e., the Validator)

- return (ICV_{n-1} == ICV_rx)

As a result, a Validator can perform an integrity check on the IOAM-Data-Fields and report whether any modification is detected. The validation is one-step in some cases (e.g., with POT Type-0 or E2E), where only the encapsulating node updates the ICV according to the definition of this method. For other cases where transit nodes also update the ICV (e.g., with Trace Option-Types), a Validator **MUST** be able to identify these transit nodes to look up their respective keys. For that, a unique identifier of the node, such as the "node_id" ([RFC9197], Section 4.4.2) for Trace Option-Types, **MUST** be included in IOAM-Data-Fields. Regardless of the Option-Type, the Nonce field allows the encapsulating node to be identified (Section 5.1). Details on how the mapping between those identifiers and keys is implemented on a Validator are out of scope.

6. IANA Considerations

6.1. IOAM Option-Types

IANA is requested to add the following new code points in the "IOAM Option-Type" registry available at [\[IANA-IOAM\]](#):

Code Point: (suggested) 64

Name: IOAM Integrity-Protected Pre-allocated Trace Option-Type

Description: Pre-allocated Trace with Integrity Protection

Reference: This document, [Section 4](#)

Code Point: (suggested) 65

Name: IOAM Integrity-Protected Incremental Trace Option-Type

Description: Incremental Trace with Integrity Protection

Reference: This document, [Section 4](#)

Code Point: (suggested) 66

Name: IOAM Integrity-Protected POT Option-Type

Description: POT with Integrity Protection

Reference: This document, [Section 4](#)

Code Point: (suggested) 67

Name: IOAM Integrity-Protected E2E Option-Type

Description: E2E with Integrity Protection

Reference: This document, [Section 4](#)

New IOAM Integrity-Protected Option-Types that intend to use the Integrity Protection Method defined in this document will update the "IOAM Integrity Protection Methods" registry ([Section 6.2](#)), more specifically the "Protected Header Fields" column of this Integrity Protection Method, to specify the list of corresponding Option-Type header fields that participate in the integrity protection of IOAM-Data-Fields. [Section 5.2](#) discusses the motivations and choices for protecting the integrity of Option-Type header fields in addition to IOAM-Data-Fields.

6.2. IOAM Integrity Protection Methods

IANA is requested to define a new registry named "IOAM Integrity Protection Methods", under the "In Situ OAM (IOAM)" registry group [IANA-IOAM].

This new registry defines 256 code points to identify different IOAM Integrity Protection Methods. The following initial code points are defined:

ID	Description	Protected Header Fields	Reference
0x00	AES-GMAC, 16-octet (full) Authentication Tag, 12-octet Initialization Vector.	Pre-allocated Trace and Incremental Trace: - Namespace-ID (mask = 0xffff) - NodeLen + Flags + RemainingLen (mask = 0xfb00) - IOAM Trace-Type (mask = 0xffffffff) - Reserved (mask = 0x00) POT: - Namespace-ID (mask = 0xffff) - IOAM-POT-Type (mask = 0xff) - IOAM-POT-Flags (mask = 0x00) E2E: - Namespace-ID (mask = 0xffff) - IOAM-E2E-Type (mask = 0xffff)	This document, Section 5
0x01 - 0xFE	Unassigned		
0xFF	Reserved		This document

Figure 7: IOAM Integrity Protection Methods

Code points 1-254 are available for assignment via the "IETF Review" process, as per [RFC8126].

New registration requests must use the following template: the value of the requested code point, a description of the Integrity Protection Method, the list of header fields with integrity protection masks for all supported Option-Types, and a reference to the document (and, optionally, the section) that defines the new Integrity Protection Method.

7. Operational Considerations

7.1. Manageability

[[I-D.ietf-ippm-ioam-integrity-yang](#)] specifies a YANG module to manage IOAM profiles with integrity protection.

7.2. Scalability and Performance

Each node that applies the Integrity Protection Method defined in this document incurs additional per-packet processing. This overhead is increased for Validators with Integrity-Protected Option-Types that require transit node participation (e.g., Integrity-Protected Trace Option-Types).

Improper use of this method can overload nodes and result in service degradation or failure. Implementations **SHOULD** monitor relevant metrics (e.g., CPU and memory utilization) to detect abnormal behavior.

Operators deploying this method **MUST** ensure that overload conditions are avoided. This can be achieved by limiting IOAM insertion to a subset of traffic, noting that only such traffic is integrity-protected. Processing load **MAY** be distributed across multiple Validators to enable parallel validation and load sharing.

7.3. Migration

Option-Types defined in [[RFC9197](#)] and the Integrity-Protected Option-Types defined in this document are not mutually exclusive and could coexist within the same IOAM-Domain. Integrity protection is an extension to IOAM.

An implementation could support multiple IOAM-Namespaces concurrently, for example, one IOAM-Namespace using the Pre-allocated Trace Option-Type and another using the Integrity-Protected Pre-allocated Trace Option-Type.

7.4. MTU

[[RFC9197](#)] discusses MTU considerations, as IOAM data is carried within packets. Similarly, the additional space required to provide integrity protection of the IOAM-Data-Fields **MUST NOT** cause packets to exceed the applicable MTU within the IOAM-Domain and **SHOULD** minimize adverse effects on packet processing.

8. Security Considerations

[Section 3](#) provides a threat analysis of integrity-related threats in the context of IOAM.

The Integrity Protection Method defined in this document ([Section 5](#)) leverages symmetric keys and uses AES-GMAC [[AES](#)] [[NIST.800-38D](#)]. Security considerations regarding key and nonce management are discussed in [Section 5.1](#). In particular, a node **MUST NOT** reuse a nonce with the same key.

Packet reordering or duplication does not compromise the Integrity Protection Method defined in this document, as key-nonce reuse is prohibited. Reordered or duplicated packets are not processed for integrity protection and no IOAM data is inserted. This behavior depends on the replay protection window implemented by a node. Such protection window determines the tolerance for packet reordering.

A compromised transit node can remove the Integrity Protection header and replace an IOAM Integrity-Protected Option-Type with its unprotected equivalent to modify IOAM-Data-Fields and bypass validation. It can also reinitialize the Nonce and ICV to impersonate an encapsulating node. To mitigate these threats, a Validator **MUST** know all IOAM Namespace-IDs with integrity protection enabled, the corresponding encapsulating nodes, and the Option-Types they add. Integrity protection **MUST** be applied to the entire IOAM set for a given Namespace-ID. Implementation details are out of scope. As noted in [Section 3.6](#), a compromised transit node can remove IOAM Option-Types; mitigation is out of scope and **SHOULD** be provided by the protocol that encapsulates IOAM.

A compromised Validator can forge or modify IOAM-Data-Fields and produce incorrect validation results. As a trusted entity, such behavior cannot be prevented by this Integrity Protection Method. Compromised encapsulating or transit nodes can forge or drop packets but cannot impersonate other IOAM nodes or modify integrity-protected IOAM-Data-Fields without detection by a Validator. A transit node's ability to forge data is limited, as validation includes the encapsulating node's ICV.

The Integrity Protection Method defined in this document is intended for Intra-IOAM-Domain use cases (i.e., no confidentiality, integrity protection only). For Inter-IOAM-Domain use cases, operators can use IPSec to securely transfer IOAM-Data-Fields between IOAM-Domains.

9. Acknowledgements

The authors would like to thank Santhosh N, Rakesh Kandula, Saiprasad Muchala, Al Morton, Greg Mirsky, Benjamin Kaduk, Mehmet Beyaz, Martin Duke, Tianran Zhou, Giuseppe Fioccola, Mohamed Boucadair, and Yoshifumi Nishida for their comments and advice.

10. References

10.1. Normative References

- [AES] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)", FIPS PUB 197, 2001, <<http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>>.

- [NIST.800-38D] National Institute of Standards and Technology, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D, 2007, <<http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC9197] Brockners, F., Ed., Bhandari, S., Ed., and T. Mizrahi, Ed., "Data Fields for In Situ Operations, Administration, and Maintenance (IOAM)", RFC 9197, DOI 10.17487/RFC9197, May 2022, <<https://www.rfc-editor.org/info/rfc9197>>.

10.2. Informative References

- [I-D.ietf-ippm-ioam-integrity-yang] Iurman, J. and T. Zhou, "A YANG Data Model for In Situ Operations, Administration, and Maintenance (IOAM) Integrity-Protected Options", Work in Progress, Internet-Draft, draft-ietf-ippm-ioam-integrity-yang-08, 8 May 2026, <<https://datatracker.ietf.org/doc/html/draft-ietf-ippm-ioam-integrity-yang-08>>.
- [I-D.ietf-opsawg-oam-characterization] Pignataro, C., Farrel, A., and T. Mizrahi, "Guidelines for Characterizing the Term "OAM"", Work in Progress, Internet-Draft, draft-ietf-opsawg-oam-characterization-17, 28 January 2026, <<https://datatracker.ietf.org/doc/html/draft-ietf-opsawg-oam-characterization-17>>.
- [IANA-IOAM] IANA, "In Situ OAM (IOAM)", <<https://www.iana.org/assignments/ioam/ioam.xhtml>>.
- [RFC7276] Mizrahi, T., Sprecher, N., Bellagamba, E., and Y. Weingarten, "An Overview of Operations, Administration, and Maintenance (OAM) Tools", RFC 7276, DOI 10.17487/RFC7276, June 2014, <<https://www.rfc-editor.org/info/rfc7276>>.
- [RFC7384] Mizrahi, T., "Security Requirements of Time Protocols in Packet Switched Networks", RFC 7384, DOI 10.17487/RFC7384, October 2014, <<https://www.rfc-editor.org/info/rfc7384>>.
- [RFC9055] Grossman, E., Ed., Mizrahi, T., and A. Hacker, "Deterministic Networking (DetNet) Security Considerations", RFC 9055, DOI 10.17487/RFC9055, June 2021, <<https://www.rfc-editor.org/info/rfc9055>>.

- [RFC9322] Mizrahi, T., Brockners, F., Bhandari, S., Gafni, B., and M. Spiegel, "In Situ Operations, Administration, and Maintenance (IOAM) Loopback and Active Flags", RFC 9322, DOI 10.17487/RFC9322, November 2022, <<https://www.rfc-editor.org/info/rfc9322>>.
- [RFC9326] Song, H., Gafni, B., Brockners, F., Bhandari, S., and T. Mizrahi, "In Situ Operations, Administration, and Maintenance (IOAM) Direct Exporting", RFC 9326, DOI 10.17487/RFC9326, November 2022, <<https://www.rfc-editor.org/info/rfc9326>>.

Authors' Addresses

Frank Brockners

Cisco Systems, Inc.
Hansaallee 249, 3rd Floor
40549 DUESSELDORF
Germany
Email: fbrockne@cisco.com

Shwetha Bhandari

Databricks
Angkor West Building, Bagmane Capital Tech Park, Ferns City
Doddanekkundi, Mahadevpura, Bengaluru, Karnataka 560048
India
Email: shwetha.bhandari@databricks.com

Tal Mizrahi

Huawei
8-2 Matam
Haifa 3190501
Israel
Email: tal.mizrahi.phd@gmail.com

Justin Iurman

University of Liege
10, Allee de la decouverte (B28)
4000 Sart-Tilman
Belgium
Email: justin.iurman@uliege.be

BIBLIOGRAPHY

- [1] J. ABLEY, P. SAVOLA, AND G. NEVILLE-NEIL, *Deprecation of Type 0 Routing Headers in IPv6*, RFC 5095, Internet Engineering Task Force, December 2007.
- [2] T. ALBAKOUR, O. GASSER, R. BEVERLY, AND G. SMARAGDAKIS, *Third Time's Not a Charm: Exploiting SNMPv3 for Router Fingerprinting*, in Proc. ACM Internet Measurement Conference (IMC), November 2021.
- [3] T. ALBAKOUR, O. GASSER, R. BEVERLY, AND G. SMARAGDAKIS, *Illuminating Router Vendor Diversity Within Providers and Along Network Paths Geolocation*, in Proc. ACM Internet Measurement Conference (IMC), October 2023.
- [4] Z. ALI, R. GANDHI, C. FILSFILS, F. BROCKNERS, N. NAINAR, C. PIGNATARO, C. LI, M. CHEN, AND G. DAWRA, *Segment Routing Header encapsulation for In-situ OAM Data*, Internet Draft (Work in Progress) draft-ali-spring-ioam-srv6-06, July 2022.
- [5] G. ALMES, S. KALIDINDI, M. ZEKAUSKAS, AND A. MORTON, *A One-Way Delay Metric for IP Performance Metrics (IPPM)*, RFC 7679, Internet Engineering Task Force, January 2016.
- [6] G. ALMES, S. KALIDINDI, M. ZEKAUSKAS, AND A. MORTON, *A One-Way Loss Metric for IP Performance Metrics (IPPM)*, RFC 7680, Internet Engineering Task Force, January 2016.
- [7] P. ALVAREZ, F. OPREA, AND J. RULA, *Rate-Limiting of IPv6 Traceroute is Widespread: Measurements and Mitigations*.
<https://www.ietf.org/proceedings/99/slides/slides-99-maprg-rate-limiting-of-ipv6-traceroutes-is-widespread-measurements-and-mitigations-01.pdf>, July 2017.
[Last Accessed: June 2nd, 2025].
- [8] S. AMANTE, B. CARPENTER, S. JIANG, AND J. RAJAHALME, *IPv6 Flow Label Specification*, RFC 6437, Internet Engineering Task Force, November 2011.
- [9] APNIC LABS, *IPv6 Fragmentation Drop Rate World Map*.
<https://stats.labs.apnic.net/v6frag>.

[Last Accessed: April, 24th 2025].

- [10] R. ATKINSON AND S. BHATTI, *IPv6 Nonce Destination Option for the Identifier-Locator Network Protocol for IPv6 (ILNPv6)*, RFC 6744, Internet Engineering Task Force, November 2012.
- [11] A. ATLASIS, *The impact of extension headers on IPv6 access control lists real life use cases*, Heidelberg, Germany, 2016.
- [12] B. AUGUSTIN, X. CUVELLIER, B. ORGOGOZO, F. VIGER, T. FRIEDMAN, M. LATAPY, C. MAGNIEN, AND R. TEIXEIRA, *Avoiding Traceroute Anomalies with Paris Traceroute*, in Proc. ACM Internet Measurement Conference (IMC), October 2006.
- [13] J.-P. AUMASSON AND D. J. BERNSTEIN, *SipHash: A Fast Short-Input PRF*, in Proc. International Conference on Cryptology in India (INFOCRYPT), December 2012.
- [14] J. BARDINELLI, Y. ZHANG, J. SU, L. HUANG, A. PARILLA, R. JARVI, S. KULKARNI, AND W. ZHANG, *hyDNS: Acceleration of DNS Through Kernel Space Resolution*, in Proc. ACM SIGCOMM, eBPF workshop, August 2024.
- [15] A. BASHANDY, C. FILSFILS, S. PREVIDI, B. DECRAENE, S. LITKOWSKI, AND R. SHAKIR, *Segment Routing with the MPLS Data Plane*, RFC 8660, Internet Engineering Task Force, December 2019.
- [16] C. BERNARDOS, A. RAHMAN, J. ZUNIGA, L. CONTRERAS, P. ARANDA, AND P. LYNCH, *Network Virtualization Research Challenges*, RFC 8568, Internet Engineering Task Force, April 2019.
- [17] G. BERTIN, *XDP in practice: integrating XDP into our DDoS mitigation pipeline*, in NetDev conference, NetDev 2.1, April 2017.
- [18] S. BHANDARI AND F. BROCKNERS, *IPv6 Options for In Situ Operations, Administration, and Maintenance (IOAM)*, RFC 9486, Internet Engineering Task Force, September 2023.
- [19] R. BHATIA, *Inside Netflix's Architecture: How It Handles Billions of Views Seamlessly*. <https://rockybhatia.substack.com/p/inside-netflixs-architecture-how>. [Last Accessed: October 6th, 2025].
- [20] M. BJORKLUND, *The YANG 1.1 Data Modeling Language*, RFC 7950, Internet Engineering Task Force, August 2016.
- [21] R. BONICA, *Deprecation of the IPv6 Router Alert Option for New Protocols*, RFC 9805, Internet Engineering Task Force, June 2025.

-
- [22] R. BONICA, Y. KAMITE, A. ALSTON, D. HENRIQUES, AND L. JALIL, *The IPv6 Compact Routing Header (CRH)*, RFC 9631, Internet Engineering Task Force, August 2024.
- [23] D. BORKMANN, *[PATCH bpf-next v6 2/8] bpf: Add fd-based tcx multi-prog infra with link support*.
<https://lore.kernel.org/all/20230719140858.13224-3-daniel@iogearbox.net/>.
[Last Accessed: August 9th, 2025].
- [24] D. BORMAN, S. DEERING, AND R. HINDEN, *IPv6 Jumbograms*, RFC 2675, Internet Engineering Task Force, August 1999.
- [25] T. BÖTTGER, F. CUADRADO, G. TYSON, I. CASTRO, AND S. UHLIG, *Open Connect Everywhere: A Glimpse at the Internet Ecosystem Through the Lens of the Netflix CDN*, ACM SIGCOMM Computer Communication Review, vol. 48, no. 1, pp. 28–34, January 2018.
- [26] V. BRENA-MEDINA, *University of Bristol Thesis Template*.
<https://fr.overleaf.com/latex/templates/university-of-bristol-thesis-template/kzqrivyxxcdm/>.
[Last Accessed: May, 12th 2025].
- [27] F. BROCKNERS AND S. BHANDARI, *Network Service Header (NSH) Encapsulation for In Situ OAM (IOAM) Data*, RFC 9452, Internet Engineering Task Force, August 2023.
- [28] F. BROCKNERS, S. BHANDARI, D. BERNIER, AND T. MIZRAHI, *In Situ Operations, Administration, and Maintenance (IOAM) Deployment*, RFC 9378, Internet Engineering Task Force, April 2023.
- [29] F. BROCKNERS, S. BHANDARI, V. GOVINDAN, C. PIGNATARO, H. GREDLER, J. LEDDY, S. YOUSELL, T. MIZRAHI, A. KFIR, B. GAFNI, P. LAPUKHOV, AND M. SPIEGEL, *VXLAN-GPE Encapsulation for In Situ OAM (IOAM) Data*, Internet Draft (Work in Progress) draft-brockners-ippm-ioam-vxlan-gpe-05, March 2024.
- [30] F. BROCKNERS, S. BHANDARI, V. GOVINDAN, C. PIGNATARO, N. NAINAR, H. GREDLER, J. LEDDY, S. YOEELL, T. MIZRAHI, P. LAPUKHOV, B. GAFNI, A. KFIR, AND M. SPIEGEL, *Geneve encapsulation for In-situ OAM Data*, Internet Draft (Work in Progress) draft-brockners-ippm-ioam-geneve-05, November 2019.
- [31] F. BROCKNERS, S. BHANDARI, AND T. MIZRAHI, *Data Fields for In Situ Operations, Administration, and Maintenance (IOAM)*, RFC 9197, Internet Engineering Task Force, May 2022.
- [32] F. BROCKNERS, S. BHANDARI, T. MIZRAHI, S. DARA, AND S. YOEELL, *Proof of Transit*, Internet Draft (Work in Progress) draft-ietf-sfc-proof-of-transit-08, Internet Engineering Task Force, October 2020.

BIBLIOGRAPHY

- [33] F. BROCKNERS, S. BHANDARI, T. MIZRAHI, AND J. IURMAN, *Integrity of In-situ OAM Data Fields*, Internet Draft (Work in Progress) draft-ietf-ippm-ioam-data-integrity-08, Internet Engineering Task Force, April 2024.
- [34] F. BROCKNERS, S. BHANDARI, T. MIZRAHI, AND J. IURMAN, *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, May 2026.
- [35] CAIDA, *CAIDA's IPv4 and IPv6 AS Core: Visualizing IPv4 and IPv6 Internet Topology at a Macroscopic Scale in 2020*.
<https://www.caida.org/projects/as-core/2020/>, 2020.
[Last Accessed: December 23rd, 2025].
- [36] R. CALLON, *Use of OSI IS-IS for Routing in TCP/IP and Dual Environments*, RFC 1195, Internet Engineering Task Force, December 1990.
- [37] CAMBRIDGE DICTIONARY, *MONITORING – English meaning*.
<https://dictionary.cambridge.org/dictionary/english/monitoring>.
[Last Accessed: September 4th, 2025].
- [38] CAMBRIDGE DICTIONARY, *OBSERVE – English meaning*.
<https://dictionary.cambridge.org/dictionary/english/observe>.
[Last Accessed: September 4th, 2025].
- [39] CAMBRIDGE DICTIONARY, *TELEMETRY – English meaning*.
<https://dictionary.cambridge.org/dictionary/english/telemetry>.
[Last Accessed: July 4th, 2025].
- [40] B. CARPENTER AND S. JIANG, *Transmission and Processing of IPv6 Extension Headers*, RFC 7045, Internet Engineering Task Force, December 2013.
- [41] B. CARPENTER AND B. LIU, *Limited Domains and Internet Protocols*, RFC 8799, Internet Engineering Task Force, July 2020.
- [42] CENTREON, *Centreon – IT Infrastructure Monitoring Software*.
<https://www.centreon.com>.
[Last Accessed: September 5th, 2025].
- [43] Y. CHEN, S. JAIN, V. K. ADHIKARI, Z. L. ZHANG, AND K. XU, *A First Look at Inter-Data Center Traffic Characteristics via Yahoo! Datasets*, in Proc. IEEE INFOCOM, April 2011.

-
- [44] T. CHOWN, J. LOUGHNEY, AND T. WINTERS, *IPv6 Node Requirements*, RFC 8504, Internet Engineering Task Force, January 2019.
- [45] K. CHU, J. SU, Y. ZHANG, C. ZHAO, Y. YANG, Y. ZHENG, S. LIN, S. ZHAO, AND W. ZHANG, *Unlocking Fine-Grained Tracing for Distributed LLM Inference with eBPF*, in Proc. ACM SIGCOMM, eBPF workshop, September 2025.
- [46] CISCO, *In-Situ OAM (IOAM)*.
<https://github.com/CiscoDevNet/iOAM/blob/master/images/>.
[Last Accessed: October 21st, 2025].
- [47] CISCO, *M-Anycast*.
<https://github.com/CiscoDevNet/iOAM/tree/master/M-Anycast>.
[Last Accessed: October 21st, 2025].
- [48] CISCO, *TRex: Realistic Traffic Generator*.
<https://trex-tgn.cisco.com>.
[Last Accessed: May, 12th 2025].
- [49] CISCO, *IPv6 Extension Headers Review and Considerations*.
https://www.cisco.com/en/US/technologies/tk648/tk872/technologies_white_paper0900aecd8054d37d.html, October 2006.
[Last Accessed: May, 15th 2025].
- [50] CISCO SYSTEMS, *IPv6 Network Management Configuration Guide, Cisco IOS Release 15M&T*.
https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipv6_nman/configuration/15-mt/ipv6n-15-mt-book/ioam-ipv6.html, 2025.
[Last Accessed: November 17th, 2025].
- [51] B. CLAISE, *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information*, RFC 5101, Internet Engineering Task Force, January 2008.
- [52] B. CLAISE, B. TRAMMELL, AND P. AITKEN, *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information*, RFC 7011, Internet Engineering Task Force, September 2013.
- [53] R. COLTUN, D. FERGUSON, J. MOY, AND A. LINDEM, *OSPF for IPv6*, RFC 5340, Internet Engineering Task Force, July 2008.
- [54] A. CONTA AND S. DEERING, *Generic Packet Tunneling in IPv6 Specification*, RFC 2473, Internet Engineering Task Force, December 1998.

- [55] A. CONTA, S. DEERING, AND M. GUPTA, *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*, RFC 4443, Internet Engineering Task Force, March 2006.
- [56] CONTAINERLAB TEAM, *Containerlab*.
<https://containerlab.dev>.
[Last Accessed: May, 19th 2025].
- [57] J. R. CONWAY, A. LEX, AND N. GEHLENBORG, *UpSetR: an R Package for the Visualization of Intersecting Sets and their Properties*, Bioinformatics, vol. 33, no. 18, September 2017.
- [58] CREATIVE COMMONS, *Attribution 4.0 International (CC BY 4.0)*.
<https://creativecommons.org/licenses/by/4.0/deed.en>.
[Last Accessed: May, 12th 2025].
- [59] A. CUSTURA, R. SECCHI, E. BOSWELL, AND G. FAIRHURST, *Is it possible to extend IPv6?*, Computer Communications, vol. 214, pp. 90–99, January 2024.
- [60] J. DANGAARD BROUER, *XDP - eXpress Data Path*.
<https://prototype-kernel.readthedocs.io/en/latest/networking/XDP>, 2016.
[Last Accessed: August 14th, 2025].
- [61] DATADOG, *Datadog – Application Performance Monitoring (APM)*.
<https://www.datadoghq.com/product/apm>, 2025.
[Last Accessed: September 1st, 2025].
- [62] S. DEERING AND R. HINDEN, *Internet Protocol, Version 6 (IPv6) Specification*, RFC 1883, Internet Engineering Task Force, December 1995.
- [63] S. DEERING AND R. HINDEN, *Internet Protocol, Version 6 (IPv6) Specification*, RFC 2460, Internet Engineering Task Force, December 1998.
- [64] S. DEERING AND R. HINDEN, *Internet Protocol, Version 6 (IPv6) Specification*, RFC 8200, Internet Engineering Task Force, July 2017.
- [65] C. DEMICHELIS AND P. CHIMENTO, *IP Packet Delay Variation Metric for IP Performance Metrics (IPPM)*, RFC 3393, Internet Engineering Task Force, November 2002.
- [66] A. DHAMDHERE AND C. DOVROLIS, *The Internet is Flat: Modeling the Transition from a Transit Hierarchy to a Peering Mesh*, in Proc. ACM CoNEXT, December 2010.
- [67] DOCKER, *Docker: Accelerated Container Application Development*.
<https://www.docker.com>.
[Last Accessed: October 6th, 2025].

- [68] DOCKER, *Docker Compose*.
<https://docs.docker.com/compose>.
[Last Accessed: October 6th, 2025].
- [69] B. DONNET AND T. FRIEDMAN, *Internet Topology Discovery: a Survey*, IEEE Communications Surveys and Tutorials, vol. 9, no. 4, pp. 56–69, December 2007.
- [70] M. DWORKING, *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*, Special Publication 800-38D, NIST, November 2007.
- [71] DYNATRACE, *Dynatrace – Application Observability*.
<https://www.dynatrace.com/platform/application-observability>, 2025.
[Last Accessed: September 1st, 2025].
- [72] W. EDDY, *Transmission Control Protocol (TCP)*, RFC 9293, Internet Engineering Task Force, August 2022.
- [73] ELASTICSEARCH, *Application Performance Monitoring (APM) with Elastic Observability*.
<https://www.elastic.co/observability/application-performance-monitoring>, 2025.
[Last Accessed: September 1st, 2025].
- [74] N. ELKINS, M. ACKERMANN, AND A. DESHPANDE, *IPv6 Extension Header (Performance and Diagnostic Metrics (PDM) Destination Option) Testing Across the Internet*.
<https://www.iepg.org/2022-07-24-ietf114/slides-114-iepg-sessa-ipv6-extension-header-pdm-do-testing-across-the-internet-00.pdf>, July 2022.
[Last Accessed: April, 24th 2025].
- [75] N. ELKINS, M. ACKERMANN, AND D. DHODY, *Deep Dive into IPv6 Extension Header Testing*, Internet Draft (Work in Progress) draft-elkins-v6ops-eh-deepdive-fw-01, Internet Engineering Task Force, October 2022.
- [76] N. ELKINS, M. ACKERMANN, AND D. DHODY, *Deep Dive into IPv6 Extension Header Testing: Behind a CDN*, Internet Draft (Work in Progress) draft-elkins-v6ops-eh-deepdive-cdn-01, Internet Engineering Task Force, February 2023.
- [77] N. ELKINS, R. HAMILTON, AND M. ACKERMANN, *IPv6 Performance and Diagnostic Metrics (PDM) Destination Option*, RFC 8250, Internet Engineering Task Force, September 2017.
- [78] N. ELKINS, P. SINHA, AND A. DESHPANDE, *Deep Dive into IPv6 Extension Header Testing: Cloud*, Internet Draft (Work in Progress) draft-elkins-v6ops-eh-deepdive-cloud-00, Internet Engineering Task Force, July 2024.

BIBLIOGRAPHY

- [79] R. ENGER AND J. REYNOLDS, *FYI on a Network Management Tool Catalog: Tools for Monitoring and Debugging TCP/IP Internets and Interconnected Devices*, RFC 1470, Internet Engineering Task Force, June 1993.
- [80] S. EVEN AND Y. MANSOUR, *A Construction of a Cipher from a Single Pseudo-Random Permutation*, Journal of Cryptology, vol. 10, no. 3, pp. 151–161, 1997.
- [81] A. FABRE, *L4Drop: XDP DDoS Mitigations*.
<https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations>, 2018.
[Last Accessed: August 14th, 2025].
- [82] FACEBOOK, *fbtracert*.
<https://github.com/facebook/fbtracert>.
[Last Accessed: October 21st, 2025].
- [83] FACEBOOK, *UdpPinger*.
<https://github.com/facebook/UdpPinger>.
[Last Accessed: October 21st, 2025].
- [84] G. FAIRHURST, *[IPv6]EH limits*.
https://mailarchive.ietf.org/arch/msg/ipv6/J_9vPNzyllNAudXwJdqdHwgzT8A/.
[Last Accessed: December 4th, 2025].
- [85] D. FARINACCI, F. MAINO, V. FULLER, AND A. CABELLOS, *Locator/ID Separation Protocol (LISP) Control Plane*, RFC 9301, Internet Engineering Task Force, October 2022.
- [86] A. FARREL AND R. BONICA, *Segment Routing: Cutting Through the Hype and Finding the IETF's Innovative Nugget of Gold*, IETF Journal, vol. 13, no. 1, July 2017.
- [87] A. FENG, P. FRANCOIS, S. FRENOT, T. GRAF, W. DU, AND P. LUCENTE, *Daisy: Practical Anomaly Detection in large BGP/MPLS and BGP/SRv6 VPN Networks*, in Proceedings of the 2023 Applied Networking Research Workshop, ANRW '23, ACM, 2023.
- [88] B. FENNER, *Experimental Values in IPv4, IPv6, ICMPv4, ICMPv6, UDP, and TCP Headers*, RFC 4727, Internet Engineering Task Force, November 2006.
- [89] C. FILSFILS, *SRv6 micro-instructions*.
<https://blog.apnic.net/2020/05/15/srv6-micro-instructions/>, 2020.
[Last Accessed: December 15th, 2025].
- [90] C. FILSFILS, P. CAMARILLO, J. LEDDY, D. VOYER, S. MATSUSHIMA, AND Z. LI, *Segment Routing over IPv6 (SRv6) Network Programming*, RFC 8986, Internet Engineering Task Force, February 2021.

-
- [91] C. FILSFILS, D. DUKES, S. PREVIDI, J. LEDDY, S. MATSUSHIMA, AND D. VOYER, *IPv6 Segment Routing Header (SRH)*, RFC 8754, Internet Engineering Task Force, March 2020.
 - [92] C. FILSFILS, S. PREVIDI, L. GINSBERG, B. DECRAENE, S. LITKOWSKI, AND R. SHAKIR, *Segment Routing Architecture*, RFC 8402, Internet Engineering Task Force, July 2018.
 - [93] C. FILSFILS, K. TALAULIKAR, D. VOYER, A. BOGDANOV, AND P. MATTES, *Segment Routing Policy Architecture*, RFC 9256, Internet Engineering Task Force, July 2022.
 - [94] G. FIOCCOLA, A. CAPELLO, M. COCIGLIO, L. CASTALDELLI, M. CHEN, L. ZHENG, G. MIRSKY, AND T. MIZRAHI, *Alternate-Marking Method for Passive and Hybrid Performance Monitoring*, RFC 8321, Internet Engineering Task Force, January 2018.
 - [95] G. FIOCCOLA, M. COCIGLIO, G. MIRSKY, T. MIZRAHI, AND T. ZHOU, *Alternate-Marking Method*, RFC 9341, Internet Engineering Task Force, December 2022.
 - [96] G. FIOCCOLA, M. COCIGLIO, A. SAPIO, AND R. SISTO, *Multipoint Alternate-Marking Method for Passive and Hybrid Performance Monitoring*, RFC 8889, Internet Engineering Task Force, August 2020.
 - [97] G. FIOCCOLA, M. COCIGLIO, A. SAPIO, R. SISTO, AND T. ZHOU, *Clustered Alternate-Marking Method*, RFC 9342, Internet Engineering Task Force, December 2022.
 - [98] G. FIOCCOLA, T. ZHOU, M. COCIGLIO, F. QIN, AND R. PANG, *IPv6 Application of the Alternate-Marking Method*, RFC 9343, Internet Engineering Task Force, December 2022.
 - [99] S. FLOYD, M. ALLMAN, A. JAIN, AND P. SAROLAHTI, *Quick-Start for TCP and IP*, RFC 4782, Internet Engineering Task Force, January 2007.
 - [100] A. GAMILLA AND M. NAAGAS, *Denial of service attack: an analysis to IPv6 extension headers security nightmares*, International Journal of Electrical and Computer Engineering (IJECE), vol. 12, June 2022.
 - [101] A. GAMILLA AND M. NAAGAS, *Header of death: security implications of IPv6 extension headers to the open-source firewall*, Bulletin of Electrical Engineering and Informatics, vol. 11, no. 1, pp. 319–326, 2022.
 - [102] R. GANDHI, F. BROCKNERS, B. WEN, B. DECRAENE, AND H. SONG, *MPLS Network Action for Transporting In Situ OAM Data Fields*, Internet Draft (Work in Progress) draft-gandhi-mpls-ioam-12, March 2024.

- [103] P. GILL, M. ARLITT, Z. LI, AND A. MAHANT, *The Flattening Internet Topology: Natural Evolution, Unsightly Barnacles or Contrived Collapse?*, in Proc. Passive and Active Measurement Conference (PAM), April 2008.
- [104] M. GOFFART, *IOAM Direct Exporting (DEX)*.
<https://github.com/Advanced-Observability/ioam-direct-exporting>.
[Last Accessed: November 17th, 2025].
- [105] M. GOFFART, J. IURMAN, E. WANSART, AND B. DONNET, *Linux Kernel Support for IOAM Direct Exporting*, in Netdev 0x19, Netdev, March 2025.
- [106] F. GONT, *Security Assessment of the Internet Protocol Version 4*, RFC 6274, Internet Engineering Task Force, July 2011.
- [107] F. GONT, *Processing of IPv6 Atomic Fragments*, RFC 6946, Internet Engineering Task Force, May 2013.
- [108] F. GONT, R. ATKINSON, AND C. PIGNATARO, *Recommendations on Filtering of IPv4 Packets Containing IPv4 Options*, RFC 7126, Internet Engineering Task Force, February 2014.
- [109] F. GONT, N. HILLIARD, G. DOERING, W. KUMARI, G. HUSTON, AND W. LIU, *Operational Implications of IPv6 Packets with Extension Headers*, RFC 9098, Internet Engineering Task Force, September 2021.
- [110] F. GONT, J. LINKOVA, T. CHOWN, AND W. LIU, *Observations on the Dropping of Packets with IPv6 Extension Headers in the Real World*, RFC 7872, Internet Engineering Task Force, June 2016.
- [111] F. GONT AND W. LIU, *Recommendations on the Filtering of IPv6 Packets Containing IPv6 Extension Headers at Transit Routers*, RFC 9288, Internet Engineering Task Force, August 2022.
- [112] F. GONT, W. LIU, AND T. ANDERSON, *Generation of IPv6 Atomic Fragments Considered Harmful*, RFC 8021, Internet Engineering Task Force, January 2017.
- [113] F. GONT, V. MANRAL, AND R. BONICA, *Implications of Oversized IPv6 Header Chains*, RFC 7112, Internet Engineering Task Force, January 2014.
- [114] GOOGLE, *Introducing gRPC, a new open source HTTP/2 RPC Framework*.
<https://developers.googleblog.com/en/introducing-grpc-a-new-open-source-http2-rpc-framework>.
[Last Accessed: October 6th, 2025].

- [115] GOOGLE, *Protocol Buffers – Google’s data interchange format*.
<https://github.com/protocolbuffers/protobuf>.
[Last Accessed: October 6th, 2025].
- [116] GOOGLE, *IPv6 Statistics*.
<https://www.google.com/intl/en/ipv6/statistics.html>, 2008–2025.
[Last Accessed: April, 24th 2025].
- [117] GRAFANA LABS, *Grafana Loki OSS – Log aggregation system*.
<https://grafana.com/oss/loki>.
[Last Accessed: September 1st, 2025].
- [118] GRAFANA LABS, *Grafana Mimir OSS – Prometheus long-term storage*.
<https://grafana.com/oss/mimir>.
[Last Accessed: September 5th, 2025].
- [119] GRAFANA LABS, *Grafana Tempo OSS – Distributed tracing backend*.
<https://grafana.com/oss/tempo>.
[Last Accessed: September 5th, 2025].
- [120] GRAYLOG, *SIEM, Log Management and API Protection*.
<https://graylog.org>.
[Last Accessed: September 5th, 2025].
- [121] GRPC, *gRPC – A high performance, open source universal RPC framework*.
<https://grpc.io>.
[Last Accessed: October 6th, 2025].
- [122] J. HALPERN AND C. PIGNATARO, *Service Function Chaining (SFC) Architecture*, RFC 7665, Internet Engineering Task Force, October 2015.
- [123] K. HEDAYAT, R. KRZANOWSKI, A. MORTON, K. YUM, AND J. BABIARZ, *A Two-Way Active Measurement Protocol (TWAMP)*, RFC 5357, Internet Engineering Task Force, October 2008.
- [124] L. HENDRIKS, P. VELAN, R. SCHMIDT, P. T. DE BOER, AND A. PRAS, *Threats and Surprises Behind IPv6 Extension Headers*, in Proc. IFIP Network Traffic Measurement and Analysis (TMA), June 2017.
- [125] U. HERBERG, A. CARDENAS, T. IWAO, M. DOW, AND S. CESPEDES, *Depth-First Forwarding (DFF) in Unreliable Networks*, RFC 6971, Internet Engineering Task Force, June 2013.

BIBLIOGRAPHY

- [126] T. HERBERT, *[IPv6]draft-ietf-6man-eh-limits*.
<https://mailarchive.ietf.org/arch/msg/ipv6/jMTQJz5bAVmLyBhtgh-pmJR3MMM/>.
[Last Accessed: December 4th, 2025].
- [127] T. HERBERT, *[PATCH net-next v2 0/7] drivers: Fix drivers doing TX csum offload with EH*.
<https://lore.kernel.org/all/20240701195507.256374-1-tom@herbertland.com/>.
[Last Accessed: August 9th, 2025].
- [128] T. HERBERT, *Limits on Sending and Processing IPv6 Extension Headers*, Internet Draft (Work in Progress) draft-ietf-6man-eh-limits-19, Internet Engineering Task Force, February 2025.
- [129] T. HERBERT AND P. LAPUKHOV, *Identifier-Locator Addressing for IPv6*, Internet Draft (Work in Progress) draft-herbert-intarea-ila-01, Internet Engineering Task Force, March 2018.
- [130] R. HINDEN AND G. FAIRHURST, *IPv6 Minimum Path MTU Hop-by-Hop Option*, RFC 9268, Internet Engineering Task Force, August 2022.
- [131] R. HINDEN AND G. FAIRHURST, *IPv6 Hop-by-Hop Options Processing Procedures*, RFC 9673, Internet Engineering Task Force, October 2024.
- [132] J.-T. HINZ, V. ADDANKI, C. GYORGYI, T. JESPEN, AND S. SCHMID, *TCP's Third Eye: Leveraging eBPF for Telemetry-Powered Congestion Control*, in Proc. Workshop on eBPF and Kernel Extensions (eBPF), September 2023.
- [133] T. HOILAND-JORGENSEN, J. DANGAARD BROUER, D. BORKMANN, J. FASTABEND, T. HERBERT, D. AHERN, AND D. MILLER, *The eXpress data path: fast programmable packet processing in the operating system kernel*, in Proc. ACM CoNEXT, December 2018.
- [134] HUAWEI TECHNOLOGIES, *What Is IOAM?*
<https://info.support.huawei.com/info-finder/encyclopedia/en/IOAM.html>, 2025.
[Last Accessed: November 17th, 2025].
- [135] J. HUI AND R. KELSEY, *Multicast Protocol for Low-Power and Lossy Networks (MPL)*, RFC 7731, Internet Engineering Task Force, February 2016.
- [136] J. HUI, J.-P. VASSEUR, D. CULLER, AND V. MANRAL, *An IPv6 Routing Header for Source Routes with the Routing Protocol for Low-Power and Lossy Networks (RPL)*, RFC 6554, Internet Engineering Task Force, March 2012.
- [137] G. HUSTON, *IPv4 Address Report*.
<https://ipv4.potaroo.net>, 2013–2025.
[Last Accessed: April, 24th 2025].

- [138] G. HUSTON, *Dealing with IPv6 Fragmentation in the DNS*.
<https://blog.apnic.net/2017/08/22/dealing-ipv6-fragmentation-dns/>, August 2017.
[Last Accessed: April, 24th 2025].
- [139] G. HUSTON AND J. DAMAS, *IPv6 extension headers revisited*.
<https://blog.apnic.net/2022/10/13/ipv6-extension-headers-revisited>, October 2022.
[Last Accessed: June, 5th 2025].
- [140] G. HUSTON AND J. DAMAS, *IPv6 Fragmentation and EH behaviours*.
<https://www.potaroo.net/presentations/2022-03-20-iepg-v6frag.pdf>, March 2022.
[Last Accessed: April, 24th 2025].
- [141] IANA – INTERNET ASSIGNED NUMBERS AUTHORITY, *Internet Protocol Version 4 (IPv4) Parameters*.
<https://www.iana.org/assignments/ip-parameters/ip-parameters.xhtml>, 2018.
[Last Accessed: August 12th, 2025].
- [142] IANA – INTERNET ASSIGNED NUMBERS AUTHORITY, *In Situ OAM (IOAM)*.
<https://www.iana.org/assignments/ioam/ioam.xhtml>, 2024.
[Last Accessed: October 6th, 2025].
- [143] IANA – INTERNET ASSIGNED NUMBERS AUTHORITY, *Internet Protocol Version 6 (IPv6) Parameters*.
<https://www.iana.org/assignments/ipv6-parameters/ipv6-parameters.xhtml>, 2024.
[Last Accessed: April, 24th 2025].
- [144] IANA – INTERNET ASSIGNED NUMBERS AUTHORITY, *Performance Metrics*.
<https://www.iana.org/assignments/performance-metrics/performance-metrics.xhtml>, 2025.
[Last Accessed: September 5th, 2025].
- [145] IANA – INTERNET ASSIGNED NUMBERS AUTHORITY, *Protocol Numbers*.
<https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml>, 2025.
[Last Accessed: April, 24th 2025].
- [146] IBM, *IBM Instana Observability*.
<https://www.ibm.com/products/instana>, 2025.
[Last Accessed: September 1st, 2025].

- [147] ICINGA, *Open Source Monitoring for Complex IT Environments – Icinga*.
<https://icinga.com>.
[Last Accessed: September 5th, 2025].
- [148] IETF ADMINISTRATION LLC, IETF– *Internet Engineering Task Force*.
<https://www.ietf.org>.
[Last Accessed: August 9th, 2025].
- [149] F. INCUBATOR, *Github – Katran*.
<https://github.com/facebookincubator/katran>, 2025.
[Last Accessed: August 14th, 2025].
- [150] J. IURMAN, *Cross Layer Telemetry*.
<https://github.com/Advanced-Observability/cross-layer-telemetry>.
[Last Accessed: October 6th, 2025].
- [151] J. IURMAN, *IOAMAgent for Python3*.
<https://github.com/Advanced-Observability/ioam-agent-python/tree/clt>.
[Last Accessed: October 6th, 2025].
- [152] J. IURMAN, *IOAMAPI with Protocol Buffers v3*.
<https://github.com/Advanced-Observability/ioam-api/tree/clt>.
[Last Accessed: October 6th, 2025].
- [153] J. IURMAN, *IOAMCollector (golang interface for Jaeger)*.
<https://github.com/Advanced-Observability/ioam-collector-go-jaeger>.
[Last Accessed: October 6th, 2025].
- [154] J. IURMAN, *IPv6 IOAM: Linux Kernel implementation*.
<https://github.com/Advanced-Observability/ioam-linux-kernel>.
[Last Accessed: October 6th, 2025].
- [155] J. IURMAN, *Carrying an Identifier in IPv6 packets*, Internet Draft (Work in Progress) draft-iurman-6man-carry-identifier-00, Internet Engineering Task Force, February 2023.
- [156] J. IURMAN, *IPv6 Extension Headers injection with eBPF*.
<https://github.com/iurmanj/ebpf-ipv6-exthdr-injection>, 2023–2025.
[Last Accessed: May, 12th 2025].
- [157] J. IURMAN, F. BROCKNERS, AND B. DONNET, *Towards cross-layer telemetry*, in *Proceedings of the 2021 Applied Networking Research Workshop, ANRW '21*, Association for Computing Machinery (ACM), 2021, pp. 15–21.

- [158] J. IURMAN AND B. DONNET, *IPv6 In-Situ Operations, Administration, and Maintenance*, Software Impacts, vol. 6, November 2020.
- [159] J. IURMAN AND B. DONNET, *Cross-Layer Telemetry Support in Linux Kernel*, in Netdev 0x16, Netdev, October 2022.
- [160] J. IURMAN AND B. DONNET, *Implementing and Evaluating IOAM Integrity Protection*, in Proceedings of the 2024 Applied Networking Research Workshop, ANRW '24, Association for Computing Machinery (ACM), 2024, pp. 60–66.
- [161] J. IURMAN AND B. DONNET, *The Razor's Edge: IPv6 Extension Headers Survivability*, in Passive and Active Measurement, Springer Nature Switzerland, 2025, pp. 3–29.
- [162] J. IURMAN, B. DONNET, AND F. BROCKNERS, *Implementation of IPv6 IOAM in Linux Kernel*, in Proc. Technical Conference on Linux Networking (Netdev 0x14), July 2020.
- [163] J. IURMAN, E. VYNCKE, AND B. DONNET, *Using eBPF to inject IPv6 Extension Headers*, in Netdev 0x17, Netdev, November 2023.
- [164] J. IURMAN, E. WANSART, M. GOFFART, AND B. DONNET, *Mitigating the Double-Reallocation Issue for IPv6 Lightweight Tunnel Encapsulations in the Linux Kernel*, March 2025.
- [165] J. IURMAN AND T. ZHOU, *A YANG Data Model for In Situ Operations, Administration, and Maintenance (IOAM) Integrity-Protected Options*, Internet Draft (Work in Progress) draft-ietf-ippm-ioam-integrity-yang-08, Internet Engineering Task Force, May 2026.
- [166] O. JACOBSEN AND D. LYNCH, *A Glossary of Networking Terms*, RFC 1208, Internet Engineering Task Force, March 1991.
- [167] JAEGER, *Jaeger: open source, distributed tracing platform*.
<https://www.jaegertracing.io>, 2025.
[Last Accessed: September 1st, 2025].
- [168] J. JAEGGLI, L. COLITTI, W. KUMARI, E. VYNCKE, M. KAE0, AND T. TAYLOR, *Why Operators Filter Fragments and What It Implies*, Internet Draft (Work in Progress) draft-taylor-v6ops-fragdrop-02, Internet Engineering Task Force, December 2013.
- [169] E. JANKIEWICZ, J. LOUGHNEY, AND T. NARTEN, *IPv6 Node Requirements*, RFC 6434, Internet Engineering Task Force, December 2011.
- [170] JUSTIN IURMAN, *IOAM Integrity: Linux kernel implementation*.
<https://github.com/Advanced-Observability/ioam-integrity-linux-kernel>, 2024.
[Last Accessed: October 6th, 2025].

BIBLIOGRAPHY

- [171] JUSTIN IURMAN, *Netdev + bpf – patchwork*.
<https://patchwork.kernel.org/project/netdevbpf/list/?series=&submitter=justin+iurman&state=3&q=iproute2&archive=both&delegate=>, 2024.
[Last Accessed: November 17th, 2025].
- [172] JUSTIN IURMAN, *Wireshark – gitlab*.
<https://gitlab.com/wireshark/wireshark/-/commits/master?search=justin+iurman>, 2024.
[Last Accessed: November 17th, 2025].
- [173] JUSTIN IURMAN, *Netdev + bpf – patchwork*.
https://patchwork.kernel.org/project/netdevbpf/list/?series=&submitter=justin+iurman&state=*&q=&archive=both&delegate=, 2025.
[Last Accessed: November 17th, 2025].
- [174] JÉRÔME BAYAUX, *Master thesis: Implementing and Validating IOAM in VPP*.
<https://github.com/Advanced-Observability/ioam-vpp>, 2021.
[Last Accessed: November 17th, 2025].
- [175] D. KATZ AND D. WARD, *Bidirectional Forwarding Detection (BFD)*, RFC 5880, Internet Engineering Task Force, June 2010.
- [176] D. KATZ AND D. WARD, *Bidirectional Forwarding Detection (BFD) for IPv4 and IPv6 (Single Hop)*, RFC 5881, Internet Engineering Task Force, June 2010.
- [177] S. KENT, *IP Authentication Header*, RFC 4302, Internet Engineering Task Force, December 2005.
- [178] S. KENT, *IP Encapsulating Security Payload (ESP)*, RFC 4303, Internet Engineering Task Force, December 2005.
- [179] S. KENT AND K. SEO, *Security Architecture for the Internet Protocol*, RFC 4301, Internet Engineering Task Force, December 2005.
- [180] T. KERNEL DEVELOPMENT COMMUNITY, *struct sk_buff*.
<https://docs.kernel.org/networking/skbuff.html>, 2024.
[Last Accessed: November 17th, 2025].
- [181] M. KERRISK, *ld.so(8) – Linux manual page*.
<https://man7.org/linux/man-pages/man8/ld.so.8.html>.
[Last Accessed: October 13th, 2025].
- [182] G. KESSLER AND S. SHEPARD, *A Primer On Internet and TCP/IP Tools and Utilities*, RFC 2151, Internet Engineering Task Force, June 1997.

- [183] R. KHEMANI, *Advanced Telemetry is a Requirement in Modern Data Center Networks*.
<https://www.forbes.com/sites/forbestechcouncil/2021/09/09/advanced-telemetry-is-a-requirement-in-modern-data-center-networks/>, September 2021.
[Last Accessed: July 4th, 2025].
- [184] C. KIM, A. SIVARAMAN, N. KATTA, A. BAS, A. DIXIT, AND L. J. WOBKER, *In-Band Network Telemetry via Programmable Dataplanes*, in In Proc. ACM SIGCOMM Conference Posters and Demos, August 2015.
- [185] S. KLASSERT ET AL., *Virtual xfrm interfaces*.
<https://lwn.net/Articles/757391/>, June 2018.
[Last Accessed: November 17th, 2025].
- [186] M. KOGIAS AND R. YANG, *HEELS: A Host-Enabled eBPF-Based Load Balancing Scheme*, in Proc. Workshop on eBPF and Kernel Extensions (eBPF), September 2023.
- [187] D. KONG, X. CHEN, H. LIN, Z. ZHOU, Y. SHEN, H. LIU, Q. CHENG, X. LIU, D. ZHANG, C. WU, AND M. K. KHAN, *Toward Security-Enhanced In-band Network Telemetry in Programmable Networks*, IEEE Transactions on Network and Service Management, 2024.
- [188] D. KONG, Z. ZHOU, Y. SHEN, X. CHEN, Q. CHENG, D. ZHANG, AND C. WU, *In-Band Network Telemetry Manipulation Attacks and Countermeasures in Programmable Networks*, in Proc. IEEE/ACM International Symposium on Quality of Service (IWQoS), June 2023.
- [189] S. KOUTSTAAL, *Bright future of SRv6*.
<https://www.ericsson.com/en/blog/2023/5/bright-future-of-srv6>, 2023.
[Last Accessed: December 15th, 2025].
- [190] S. KRISHNAN, A. KAVANAGH, B. VARGA, S. OOGHE, AND E. NORDMARK, *The Line-Identification Option*, RFC 6788, Internet Engineering Task Force, November 2012.
- [191] S. KRISHNAN, J. WOODYATT, E. KLINE, J. HOAGLAND, AND M. BHATIA, *A Uniform Format for IPv6 Extension Headers*, RFC 6564, Internet Engineering Task Force, April 2012.
- [192] M. KUHLEWIND, B. TRAMMELL, T. BÜHLER, G. FAIRHURST, AND V. GURBANI, *Challenges in Network Management of Encrypted Traffic*, 2018.
- [193] R. LÉAS, J. IURMAN, E. VYNCKE, AND B. DONNET, *Measuring IPv6 Extension Headers Survivability with JAMES*, in Proc. ACM Internet Measurement Conference (IMC), Poster Session, October 2022.

- [194] B. LIN, L. ZHANG, Y. GUO, H. ZHANG, AND Y. FANG, *Research on Security Protection Evasion Mechanism Based on IPv6 Fragment Headers*, in 2024 IEEE 49th Conference on Local Computer Networks (LCN), 2024, pp. 1–9.
- [195] LINUX, *ping6(8) - Linux man page*.
<https://linux.die.net/man/8/ping6>.
[Last Accessed: May, 19th 2025].
- [196] LINUX FOUNDATION, *What is the Vector Packet Processor (vpp)*.
<https://fd.io/docs/vpp/master>, 2025.
[Last Accessed: November 17th, 2025].
- [197] LWN.NET, *net, sched: add clsact qdisc*.
<https://lwn.net/Articles/671458/>, January 2016.
[Last Accessed: May, 12th 2025].
- [198] C. LYNN, *Endpoint Identifier Destination Option*, Internet Draft (Work in Progress) draft-ietf-nimrod-eid-00, Internet Engineering Task Force, November 1995.
- [199] J. MACKER, *Simplified Multicast Forwarding*, RFC 6621, Internet Engineering Task Force, May 2012.
- [200] L. MAN PAGES, *netlink(7) – Linux manual page*.
<https://man7.org/linux/man-pages/man7/netlink.7.html>.
[Last Accessed: November 17th, 2025].
- [201] E. MARECHAL AND B. DONNET, *Network fingerprinting: Routers under attack*, in Proc. International Workshop on Traffic Measurements for Cybersecurity (WTMC), September 2020.
- [202] D. MCGREW AND J. VIEGA, *The Use of Galois Message Authentication Code (GMAC) in IPsec ESP and AH*, RFC 4543, Internet Engineering Task Force, May 2006.
- [203] MICROSOFT, *eBPF for Windows*.
<https://microsoft.github.io/eBPF-for-Windows>.
[Last Accessed: August 14th, 2025].
- [204] MICROSOFT, *Github – eBPF for Windows*.
<https://github.com/microsoft/eBPF-for-Windows>, 2025.
[Last Accessed: August 14th, 2025].
- [205] MICROSOFT, *Github – XDP for Windows*.
<https://github.com/microsoft/xdp-for-windows>, 2025.
[Last Accessed: August 14th, 2025].

- [206] MICROSOFT, *Welcome to Windows*.
<https://www.microsoft.com/en-us/windows>, 2025.
[Last Accessed: August 14th, 2025].
- [207] X. MIN, Z. ZHANG, Y. LIU, N. NAINAR, AND C. PIGNATARO, *BIER Encapsulation for IOAM Data*, Internet Draft (Work in Progress) draft-xzlnp-bier-ioam-08, July 2024.
- [208] G. MIRSKY, G. JUN, H. NYDELL, AND R. FOOTE, *Simple Two-Way Active Measurement Protocol*, RFC 8762, Internet Engineering Task Force, March 2020.
- [209] G. MIRSKY, W. LINGQIANG, G. ZHUI, H. SONG, AND P. THUBERT, *Hybrid Two-Step Performance Measurement Method*, Internet Draft (Work in Progress) draft-ietf-ippm-hybrid-two-step-05, Internet Engineering Task Force, August 2025.
- [210] T. MIZRAHI, F. BROCKNERS, S. BHANDARI, B. GAFNI, AND M. SPIEGEL, *In Situ Operations, Administration, and Maintenance (IOAM) Loopback and Active Flags*, RFC 9322, Internet Engineering Task Force, November 2022.
- [211] T. MIZRAHI, F. BROCKNERS, A. CLEMM, J. IURMAN, S. BHANDARI, AND T. ZHOU, *In Situ Operations, Administration, and Maintenance (IOAM) Template Option*, Internet Draft (Work in Progress) draft-mbci-ippm-ioam-template-option-00, Internet Engineering Task Force, July 2025.
- [212] T. MIZRAHI, J. IURMAN, AND F. BROCKNERS, *In Situ OAM Profile for the Linux Kernel Implementation*, Internet Draft (Work in Progress) draft-mizrahi-ippm-ioam-linux-profile-02, January 2022.
- [213] T. MIZRAHI, N. SPRECHER, E. BELLAGAMBA, AND Y. WEINGARTEN, *An Overview of Operations, Administration, and Maintenance (OAM) Tools*, RFC 7276, Internet Engineering Task Force, June 2014.
- [214] T. MIZRAHI, T. ZHOU, S. BELKAR, R. COHEN, AND J. IURMAN, *Internet Control Message Protocol (ICMPv6) Loopback*, Internet Draft (Work in Progress) draft-mcb-6man-icmpv6-loopback-01, June 2024.
- [215] Q. MONNET, *Fosdem '24 – Packet, where are you?*
https://archive.fosdem.org/2024/events/attachments/fosdem-2024-3425-packet-where-are-you-track-in-the-stack-with-pwru/slides/22915/FOSDEM2024-qmonnet-pwru_Y9an8XJ.pdf, 2024.
[Last Accessed: August 14th, 2025].
- [216] J. MOREAU, *Netflix Crashes as 'Stranger Things' Finale Premieres, Second Time This Season*.

BIBLIOGRAPHY

- <https://variety.com/2025/tv/news/netflix-down-crashes-stranger-things-finale-1236621055/>, 2025.
[Last Accessed: January 2nd, 2026].
- [217] A. MORTON, *Active and Passive Metrics and Methods (with Hybrid Types In-Between)*, RFC 7799, Internet Engineering Task Force, May 2016.
- [218] R. MOSKOWITZ, T. HEER, P. JOKELA, AND T. HENDERSON, *Host Identity Protocol Version 2 (HIPv2)*, RFC 7401, Internet Engineering Task Force, April 2015.
- [219] K. MUTHUKRISHNAN AND A. MALIS, *A Core MPLS IP VPN Architecture*, RFC 2917, Internet Engineering Task Force, September 2000.
- [220] NAGIOS ENTERPRISES, *Nagios – Open Source Monitoring and Network Management*.
<https://www.nagios.org>.
[Last Accessed: September 5th, 2025].
- [221] T. NARTEN, *Assigning Experimental and Testing Numbers Considered Useful*, RFC 3692, Internet Engineering Task Force, January 2004.
- [222] NETCAT PROJECT, *The GNU Netcat project*.
<https://netcat.sourceforge.net>.
[Last Accessed: May, 19th 2025].
- [223] NETDATA, *Open Source Monitoring – Netdata*.
<https://www.netdata.cloud/open-source>.
[Last Accessed: September 5th, 2025].
- [224] NEW RELIC, *New Relic – Application Performance Monitoring (APM)*.
<https://newrelic.com/lp/apm>, 2025.
[Last Accessed: September 1st, 2025].
- [225] K. NICHOLS, S. BLAKE, F. BAKER, AND D. BLACK, *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*, RFC 2474, Internet Engineering Task Force, December 1998.
- [226] Y. NOFEL, *From Counters to Telemetry: A Survey of Programmable Network-Wide Monitoring*, Network, vol. 5, 2025.
- [227] E. NORDMARK AND M. BAGNULO, *Shim6: Level 3 Multihoming Shim Protocol for IPv6*, RFC 5533, Internet Engineering Task Force, June 2009.
- [228] NTOP, *Open Source packet capture framework*.
https://www.ntop.org/products/packet-capture/pf_ring.
[Last Accessed: October 13th, 2025].

- [229] OPENTELEMETRY, *Agent – OpenTelemetry*.
<https://opentelemetry.io/docs/collector/deployment/agent>, 2024.
[Last Accessed: September 1st, 2025].
- [230] OPENTELEMETRY, *Observability primer – OpenTelemetry*.
<https://opentelemetry.io/docs/concepts/observability-primer>, 2024.
[Last Accessed: September 1st, 2025].
- [231] OPENTELEMETRY, *Github – OpenTelemetry – cncf*.
<https://github.com/open-telemetry>, 2025.
[Last Accessed: August 21st, 2025].
- [232] OPENTELEMETRY, *OpenTelemetry*.
<https://opentelemetry.io>, 2025.
[Last Accessed: August 21st, 2025].
- [233] OPENZIPKIN, *OpenZipkin – A distributed tracing system*.
<https://zipkin.io>.
[Last Accessed: September 5th, 2025].
- [234] K. OUELLETTE, *IPv6 Hop-by-hop and Destination Options Forwarding In Routers*, Internet Draft (Work in Progress) draft-ouellette-v6ops-eh-router-forwarding-00, Internet Engineering Task Force, March 2024.
- [235] PAESSLER, *Network Monitoring Tool – PRTG*.
<https://www.paessler.com/monitoring/network/network-monitoring-tool>.
[Last Accessed: September 5th, 2025].
- [236] X. PAN, S. TANG, S. LIU, J. LONG, X. ZHANG, D. HU, J. QI, AND Z. ZHU, *Privacy-Preserving Multilayer In-Band Network Telemetry and Data Analytics: For Safety, Please do Not Report Plaintext Data*, Journal of Lightwave Technology, vol. 38, no. 21, pp. 5855–5866, November 2020.
- [237] C. PERKINS, D. JOHNSON, AND J. ARKKO, *Mobility Support in IPv6*, RFC 6275, Internet Engineering Task Force, July 2011.
- [238] P. PHAAL, S. PANCHEN, AND N. MCKEE, *InMon Corporation’s sFlow: A Method for Monitoring Traffic in Switched and Routed Networks*, RFC 3176, Internet Engineering Task Force, September 2001.
- [239] PHILIP HANE, *ipwhois – PyPI*.
<https://pypi.org/project/ipwhois>.
[Last Accessed: May, 30th 2025].

BIBLIOGRAPHY

- [240] C. PIGNATARO AND F. GONT, *Formally Deprecating Some IPv4 Options*, RFC 6814, Internet Engineering Task Force, November 2012.
- [241] J. POSTEL, *User Datagram Protocol*, RFC 768, Internet Engineering Task Force, August 1980.
- [242] J. POSTEL, *Internet Protocol*, RFC 791, Internet Engineering Task Force, September 1981.
- [243] PROMETHEUS, *Prometheus – Monitoring system and time series database*.
<https://prometheus.io>, 2025.
[Last Accessed: September 1st, 2025].
- [244] K. RAMAKRISHNAN, S. FLOYD, AND D. BLACK, *The Addition of Explicit Congestion Notification (ECN) to IP*, RFC 3168, Internet Engineering Task Force, September 2001.
- [245] R. RAMANATHAN AND M. STEENSTRUP, *Nimrod Functionality and Protocol Specifications, Version 1*, Internet Draft (Work in Progress) draft-ietf-nimrod-fun-pro-spec-00, Internet Engineering Task Force, March 1996.
- [246] Y. REKHTER, T. LI, AND S. HARES, *A Border Gateway Protocol 4 (BGP-4)*, RFC 4271, Internet Engineering Task Force, January 2006.
- [247] RIPE NCC, *Ripe Atlas*.
<https://atlas.ripe.net>.
[Last Accessed: May, 24th 2025].
- [248] M. ROBLES, M. RICHARDSON, AND P. THUBERT, *Using RPI Option Type, Routing Header for Source Routes, and IPv6-in-IPv6 Encapsulation in the RPL Data Plane*, RFC 9008, Internet Engineering Task Force, April 2021.
- [249] R. ROOPA AND T. G., *Lightweight & flow based tunneling*.
<https://lwn.net/Articles/650778/>, July 2015.
[Last Accessed: November 17th, 2025].
- [250] E. ROSEN, A. VISWANATHAN, AND R. CALLON, *Multiprotocol Label Switching Architecture*, RFC 3031, Internet Engineering Task Force, January 2001.
- [251] J. SALIM, H. KHOSRAVI, A. KLEEN, AND A. KUZNETSOV, *Linux Netlink as an IP Services Protocol*, RFC 3549, Internet Engineering Task Force, July 2003.
- [252] SCAPY COMMUNITY, *Scapy*.
<https://scapy.net>.
[Last Accessed: May, 12th 2025].

- [253] S. SHALUNOV, B. TEITELBAUM, A. KARP, J. BOOTE, AND M. ZEKAUSKAS, *A One-way Active Measurement Protocol (OWAMP)*, RFC 4656, Internet Engineering Task Force, September 2006.
- [254] A. SHAMIR, *How to share a secret*, Commun. ACM, vol. 22, no. 11, pp. 612–613, 1979.
- [255] N. SHIROKOV AND R. DASINENI, *Open-sourcing Katran, a scalable network load balancer*. <https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer>, 2018.
[Last Accessed: August 14th, 2025].
- [256] SIGNOZ, *SigNoz – The Open Source Datadog Alternative*. <https://signoz.io>, 2025.
[Last Accessed: September 1st, 2025].
- [257] S. SINGH, *Netflix Subscribers Statistics 2025 (Active Users & Revenue)*. <https://www.demandsage.com/netflix-subscribers/>, 2025.
[Last Accessed: December 23rd, 2025].
- [258] P. SODHI, G. LIARGKOVAS, AND K. KAFFES, *Empowering machine-learning assisted kernel decisions with eBPF+ML*, in Proc. ACM SIGCOMM, eBPF workshop, September 2025.
- [259] SOLARWINDS WORLDWIDE, *Network Performance Monitor*. <https://www.solarwinds.com/network-performance-monitor>.
[Last Accessed: September 5th, 2025].
- [260] H. SONG, B. GAFNI, F. BROCKNERS, S. BHANDARI, AND T. MIZRAHI, *In Situ Operations, Administration, and Maintenance (IOAM) Direct Exporting*, RFC 9326, Internet Engineering Task Force, November 2022.
- [261] H. SONG, M. MCBRIDE, G. MIRSKY, G. MISHRA, H. ASAEDA, AND T. ZHOU, *Multicast On-Path Telemetry Using In Situ Operations, Administration, and Maintenance (IOAM)*, RFC 9630, Internet Engineering Task Force, August 2024.
- [262] H. SONG, F. QIN, P. MARTINEZ-JULIA, L. CIAVAGLIA, AND A. WANG, *Network Telemetry Framework*, RFC 9232, Internet Engineering Task Force, May 2022.
- [263] SPLUNK, *Splunk – Application Performance Monitoring*. https://www.splunk.com/en_us/products/apm-application-performance-monitoring.html, 2025.
[Last Accessed: September 1st, 2025].
- [264] M. STJOHNS, R. ATKINSON, AND G. THOMAS, *Common Architecture Label IPv6 Security Option (CALIPSO)*, RFC 5570, Internet Engineering Task Force, July 2009.

BIBLIOGRAPHY

- [265] C. TEAM, *Github – pwru (packet, where are you?)*.
<https://github.com/cilium/pwru>, 2025.
[Last Accessed: August 14th, 2025].
- [266] THE APACHE SOFTWARE FOUNDATION, *Apache SkyWalking*.
<https://skywalking.apache.org>, 2025.
[Last Accessed: September 1st, 2025].
- [267] THE LINUX DOCUMENTATION PROJECT, *Introduction to iproute2*.
<https://tldp.org/HOWTO/Adv-Routing-HOWTO/lartc.iproute2.html>.
[Last Accessed: May, 12th 2025].
- [268] THE LINUX DOCUMENTATION PROJECT, *Traffic Control HOWTO*.
<https://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html>.
[Last Accessed: May, 12th 2025].
- [269] THE LINUX FOUNDATION, *eBPF*.
<https://ebpf.io>.
[Last Accessed: May, 12th 2025].
- [270] THE LINUX FOUNDATION, *Kubernetes*.
<https://kubernetes.io>.
[Last Accessed: October 21st, 2025].
- [271] THE LINUX FOUNDATION, *Cloud Native Computing Foundation*.
<https://www.cncf.io>, 2025.
[Last Accessed: September 1st, 2025].
- [272] THE LINUX FOUNDATION, *Jaeger – CNCF*.
<https://www.cncf.io/projects/jaeger>, 2025.
[Last Accessed: September 1st, 2025].
- [273] THE LINUX FOUNDATION PROJECTS, *DPDK – The open source data plane development kit accelerating network performance*.
<https://www.dpdk.org>.
[Last Accessed: May, 12th 2025].
- [274] THE LINUX KERNEL, *drivers/net/ethernet/intel/i40e/i40e_txrx.c – Linux source code v6.9 – Bootlin Elixir Cross Referencer*.
https://elixir.bootlin.com/linux/v6.9/source/drivers/net/ethernet/intel/i40e/i40e_txrx.c#L2137.
[Last Accessed: May, 12th 2025].

- [275] THE LINUX KERNEL, *libbpf Overview – The Linux Kernel documentation*.
https://docs.kernel.org/bpf/libbpf/libbpf_overview.html.
[Last Accessed: May, 12th 2025].
- [276] THE LLVM PROJECT, *Clang: a C language family frontend for LLVM*.
<https://clang.llvm.org>.
[Last Accessed: May, 12th 2025].
- [277] THE P4.ORG APPLICATIONS WORKING GROUP, *In-band Network Telemetry (INT) Data-plane Specification – Version 2.1*.
https://p4.org/p4-spec/docs/INT_v2_1.pdf, 2020.
[Last Accessed: September 8th, 2025].
- [278] THE TCPDUMP GROUP, *Home – TCPDUMP and LIBPCAP*.
<https://www.tcpdump.org>.
[Last Accessed: May, 19th 2025].
- [279] THE TCPDUMP GROUP, *Libpcap by The Tcpdump Group – GitHub*.
<https://github.com/the-tcpdump-group/libpcap>, 2025.
[Last Accessed: September 10th, 2025].
- [280] J. TOUCH AND W. TOWNSLEY, *IP Tunnels in the Internet Architecture*, Internet Draft (Work in Progress) draft-ietf-intarea-tunnels-15, Internet Engineering Task Force, May 2025.
- [281] B. TRAMMELL, M. WELZL, R. ENGHARDT, G. FAIRHURST, M. KUHLEWIND, C. PERKINS, P. TIESEL, AND T. PAULY, *An Abstract Application Programming Interface (API) for Transport Services*, RFC 9622, Internet Engineering Task Force, January 2025.
- [282] V.-H. TRAN AND O. BONAVENTURE, *Beyond Socket Options: Towards Fully Extensible Linux Transport Stacks*, Computer Communications, vol. 1, pp. 118–138, October 2020.
- [283] TUXOLOGY, *An entertaining eBPF XDP adventure*.
<https://suchakra.files.wordpress.com/2017/05/cls-xdp1.png>.
[Last Accessed: May, 12th 2025].
- [284] UBER TECHNOLOGIES, *Uber*.
<https://www.uber.com>, 2025.
[Last Accessed: September 1st, 2025].
- [285] J. ULLRICH, K. KROMBHOLZ, H. HOBEL, A. DABROWSKI, AND E. WEIPPL, *IPv6 security: Attacks and countermeasures in a nutshell*, in 8th USENIX Workshop on Offensive Technologies (WOOT 14), 2014.

- [286] O. V.-A PADUREAN, GASSER, R. BUSH, AND A. FELDMANN, *SRv6: Is There Anybody Out There?*, in Proc. International Workshop on Traffic Measurements for Cybersecurity (WTMC), June 2022.
- [287] P. L. VENTRE, S. SALSANO, M. POLVERINI, A. CIANFRANI, A. ABDELSALAM, C. FILSFILS, P. CAMARILLO, AND F. CLAD, *Segment Routing: A Comprehensive Survey of Research Activities, Standardization Efforts, and Implementation Results*, IEEE Communications Surveys & Tutorials, vol. 23, no. 1, pp. 182–221, November 2020.
- [288] F. G. VOGT, F. RODRIGUEZ, M. C. LUIZELLI, AND C. E. ROTHENBERG, *Rethinking the In-band Network Telemetry: Towards Application and Server-Level Network Telemetry*, in Proc. ACM CoNEXT on Student Workshop 2024, 2024, pp. 21–22.
- [289] E. VYNCKE, B. DONNET, AND J. IURMAN, *Attribution of Internet Probes*, RFC 9511, Internet Engineering Task Force, November 2023.
- [290] E. VYNCKE, R. LÉAS, AND J. IURMAN, *Just Another Measurement of Extension header Survivability (JAMES)*, Internet Draft (Work in Progress) draft-vyncke-v6ops-james-02, Internet Engineering Task Force, July 2022.
- [291] R. WANG, Z. WANG, D. WANG, AND Y. LIU, *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), December 2021.
- [292] E. WANSART, *IOAMAgent*.
<https://github.com/Advanced-Observability/ioam-agent-go/tree/clt>.
[Last Accessed: October 13th, 2025].
- [293] E. WANSART, *IOAMAgent for C++*.
<https://github.com/Advanced-Observability/ioam-agent-cpp/tree/clt>.
[Last Accessed: October 13th, 2025].
- [294] B. WEIS, F. BROCKNERS, C. HILL, S. BHANDARI, V. GOVINDAN, C. PIGNATARO, N. NAINAR, H. GREDLER, J. LEDDY, S. YOUELL, T. MIZRAHI, A. KFIR, B. GAFNI, P. LAPUKHOV, AND M. SPIEGEL, *EtherType Protocol Identification of In-situ OAM Data*, Internet Draft (Work in Progress) draft-weis-ippm-ioam-eth-05, February 2022.
- [295] WIKIMEDIA, *observability – Wiktionary, the free dictionary*.
<https://en.wiktionary.org/wiki/observability>.
[Last Accessed: September 4th, 2025].
- [296] WIKIPEDIA, *History of the Internet*.
https://en.wikipedia.org/wiki/History_of_the_Internet, 2025.
[Last Accessed: December 23rd, 2025].

- [297] T. WINTER, P. THUBERT, A. BRANDT, J. HUI, R. KELSEY, P. LEVIS, K. PISTER, R. STRUIK, J. VASSEUR, AND R. ALEXANDER, *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*, RFC 6550, Internet Engineering Task Force, March 2012.
- [298] WIRESHARK FOUNDATION, *Wireshark – Go Deep*.
<https://www.wireshark.org>.
[Last Accessed: September 5th, 2025].
- [299] L. WUSTRICH, M. SCHACHERBAUER, M. BUDEUS, D. FREIHERR VON KUNBBERG, S. GALLENMULLER, M.-O. PAHL, AND G. CARLE, *Network Profiles for Detecting Application-Characteristic Behavior using Linux eBPF*, in Proc. Workshop on eBPF and Kernel Extensions (eBPF), September 2023.
- [300] M. XHONNEUX, F. DUCHENE, AND O. BONAVENTURE, *Leveraging eBPF for Programmable Network Functions with IPv6 Segment Routing*, in Proc. ACM CoNEXT, November 2018.
- [301] F. YI, *PhD Thesis: Cross-Layer Telemetry and Optimization for Real-Time Interactive Applications in 5G Networks*, PhD thesis, September 2025.
- [302] M. YU, *Network Telemetry: Towards a Top-Down Approach*, ACM SIGCOMM Computer Communication Review, vol. 49, no. 1, pp. 11–17, January 2019.
- [303] ZABBIX, *Network Monitoring – Zabbix*.
https://www.zabbix.com/network_monitoring.
[Last Accessed: September 5th, 2025].
- [304] H. ZHAO AND J. BI, *Characterizing and Analysis of the Flattening Internet Topology*, in Proc. International Symposium on Computers and Communications (ISCC), July 2013.
- [305] Y. ZHAO, G. CHENG, AND Y. TANG, *SINT: Toward a Blockchain-Based Secure In-Band Network Telemetry Architecture*, IEEE Transactions on Information Forensics and Security, vol. 18, pp. 2667–2682, April 2023.

