Internet Engineering Task Force (IETF)
Request for Comments: 8956
Updates: 8955
Category: Standards Track
ISSN: 2070-1721

C. Loibl, Ed.
next layer Telekom GmbH
R. Raszuk, Ed.
NTT Network Innovations
S. Hares, Ed.
Huawei

December 2020

Dissemination of Flow Specification Rules for IPv6

Abstract

"Dissemination of Flow Specification Rules" (RFC 8955) provides a Border Gateway Protocol (BGP) extension for the propagation of traffic flow information for the purpose of rate limiting or filtering IPv4 protocol data packets.

This document extends RFC 8955 with IPv6 functionality. It also updates RFC 8955 by changing the IANA Flow Spec Component Types registry.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8956.

Table of Contents

1. Introduction

The growing amount of IPv6 traffic in private and public networks requires the extension of tools used in IPv4-only networks to also support IPv6 data packets.

This document analyzes the differences between describing IPv6 [RFC8200] flows and those of IPv4 packets. It specifies new Border Gateway Protocol [RFC4271] encoding formats to enable "Dissemination of Flow Specification Rules" [RFC8955] for IPv6.

This specification is an extension of the base established in [RFC8955]. It only defines the delta changes required to support IPv6, while all other definitions and operation mechanisms of "Dissemination of Flow Specification Rules" will remain in the main specification and will not be repeated here.

1.1. Definitions of Terms Used in This Memo

AFI: Address Family Identifier

AS: Autonomous System

NLRI: Network Layer Reachability Information

SAFI: Subsequent Address Family Identifier

VRF: Virtual Routing and Forwarding

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. IPv6 Flow Specification Encoding in BGP

[RFC8955] defines SAFIs 133 (Dissemination of Flow Specification rules) and 134 (L3VPN Dissemination of Flow Specification rules) in order to carry the corresponding Flow Specification.

Implementations wishing to exchange IPv6 Flow Specifications MUST use BGP's Capability Advertisement facility to exchange the Multiprotocol Extension Capability Code (Code 1), as defined in [RFC4760]. The (AFI, SAFI) pair carried in the Multiprotocol Extension Capability MUST be (AFI=2, SAFI=133) for IPv6 Flow Specification rules and (AFI=2, SAFI=134) for L3VPN Dissemination of Flow Specification rules.

3. IPv6 Flow Specification Components

The encoding of each of the components begins with a Type field (1 octet) followed by a variable length parameter. The following sections define component types and parameter encodings for IPv6.

Types 4 (Port), 5 (Destination Port), 6 (Source Port), 9 (TCP Flags), 10 (Packet Length), and 11 (DSCP), as defined in [RFC8955], also apply to IPv6. Note that IANA has updated the "Flow Spec Component Types" registry in order to contain both IPv4 and IPv6 Flow Specification component type numbers in a single registry (Section 8).

3.1. Type 1 - Destination IPv6 Prefix

Encoding: <type (1 octet), length (1 octet), offset (1 octet), pattern (variable), padding (variable) >

This defines the destination prefix to match. The offset has been defined to allow for flexible matching to portions of an IPv6 address where one is required to skip over the first N bits of the address. (These bits skipped are often indicated as "don't care" bits.) This can be especially useful where part of the IPv6 address consists of an embedded IPv4 address, and matching needs to happen only on the embedded IPv4 address. The encoded pattern contains enough octets for the bits used in matching (length minus offset bits).

length: This indicates the N-th most significant bit in the address where bitwise pattern matching stops.

offset: This indicates the number of most significant address bits to skip before bitwise pattern matching starts.

pattern: This contains the matching pattern. The length of the pattern is defined by the number of bits needed for pattern matching (length minus offset).

padding: This contains the minimum number of bits required to pad the component to an octet boundary. Padding bits MUST be

0 on encoding and MUST be ignored on decoding.

If length = 0 and offset = 0, this component matches every address; otherwise, length MUST be in the range offset < length < 129 or the component is malformed.

Note: This Flow Specification component can be represented by the notation ipv6address/length if offset is 0 or ipv6address/offset-length. The ipv6address in this notation is the textual IPv6 representation of the pattern shifted to the right by the number of offset bits. See also Section 3.8.

3.2. Type 2 - Source IPv6 Prefix

Encoding: <type (1 octet), length (1 octet), offset (1 octet), pattern (variable), padding (variable) >

This defines the source prefix to match. The length, offset, pattern, and padding are the same as in Section 3.1.

3.3. Type 3 - Upper-Layer Protocol

Encoding: <type (1 octet), [numeric_op, value]+>

This contains a list of {numeric_op, value} pairs that are used to match the first Next Header value octet in IPv6 packets that is not an extension header and thus indicates that the next item in the packet is the corresponding upper-layer header (see Section 4 of [RFC8200]).

This component uses the Numeric Operator (numeric_op) described in Section 4.2.1.1 of [RFC8955]. Type 3 component values SHOULD be encoded as a single octet (numeric_op len=00).

Note: While IPv6 allows for more than one Next Header field in the packet, the main goal of the Type 3 Flow Specification component is to match on the first upper-layer IP protocol value. Therefore, the definition is limited to match only on this specific Next Header field in the packet.

3.4. Type 7 - ICMPv6 Type

Encoding: <type (1 octet), [numeric_op, value]+>

This defines a list of {numeric_op, value} pairs used to match the Type field of an ICMPv6 packet (see also Section 2.1 of [RFC4443]).

This component uses the Numeric Operator (numeric_op) described in Section 4.2.1.1 of [RFC8955]. Type 7 component values SHOULD be encoded as a single octet (numeric_op len=00).

In case of the presence of the ICMPv6 type component, only ICMPv6 packets can match the entire Flow Specification. The ICMPv6 type component, if present, never matches when the packet's upper-layer IP protocol value is not 58 (ICMPv6), if the packet is fragmented and this is not the first fragment, or if the system is unable to locate

the transport header. Different implementations may or may not be able to decode the transport header.

3.5. Type 8 - ICMPv6 Code

Encoding: <type (1 octet), [numeric_op, value]+>

This defines a list of {numeric_op, value} pairs used to match the code field of an ICMPv6 packet (see also Section 2.1 of [RFC4443]).

This component uses the Numeric Operator (numeric_op) described in Section 4.2.1.1 of [RFC8955]. Type 8 component values SHOULD be encoded as a single octet (numeric_op len=00).

In case of the presence of the ICMPv6 code component, only ICMPv6 packets can match the entire Flow Specification. The ICMPv6 code component, if present, never matches when the packet's upper-layer IP protocol value is not 58 (ICMPv6), if the packet is fragmented and this is not the first fragment, or if the system is unable to locate the transport header. Different implementations may or may not be able to decode the transport header.

3.6. Type 12 - Fragment

Encoding: <type (1 octet), [bitmask_op, bitmask]+>

This defines a list of {bitmask_op, bitmask} pairs used to match specific IP fragments.

This component uses the Bitmask Operator (bitmask_op) described in Section 4.2.1.2 of [RFC8955]. The Type 12 component bitmask MUST be encoded as a single octet bitmask (bitmask_op len=00).

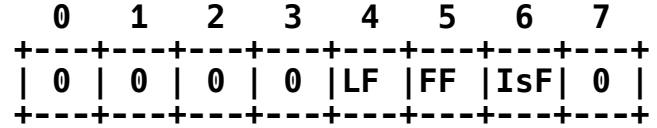

Figure 1: Fragment Bitmask Operand

Bitmask values:

IsF: Is a fragment other than the first -- match if IPv6 Fragment Header (Section 4.5 of [RFC8200]) Fragment Offset is not 0

FF: First fragment -- match if IPv6 Fragment Header (Section 4.5 of [RFC8200]) Fragment Offset is 0 AND M flag is 1

LF: Last fragment -- match if IPv6 Fragment Header (Section 4.5 of [RFC8200]) Fragment Offset is not 0 AND M flag is 0

0: MUST be set to 0 on NLRI encoding and MUST be ignored during decoding

3.7. Type 13 - Flow Label (new)

Encoding: <type (1 octet), [numeric_op, value]+>

This contains a list of {numeric_op, value} pairs that are used to match the 20-bit Flow Label IPv6 header field (Section 3 of [RFC8200]).

This component uses the Numeric Operator (numeric_op) described in Section 4.2.1.1 of [RFC8955]. Type 13 component values SHOULD be encoded as 4-octet quantities (numeric_op len=10).

3.8. Encoding Examples

3.8.1. Example 1

The following example demonstrates the prefix encoding for packets from ::1234:5678:9a00:0/64-104 to 2001:db8::/32 and upper-layer protocol tcp.

len	destination	source	ul-proto
0x12	01 20 00 20 01 0d bb	02 68 40 12 34 56 78 9a	03 81 06

Table 1

Decoded:

Value		
0x12	length	18 octets (if len<240, 1 octet)
0x01	type	Type 1 - Dest. IPv6 Prefix
0x20	length	32 bits
0x00	offset	0 bits
0x20	pattern	
0x01	pattern	
0x0d	pattern	
0xb8	pattern	(no padding needed)
0x02	type	Type 2 - Source IPv6 Prefix
0x68	length	104 bits
0x40	offset	64 bits
0x12	pattern	
0x34	pattern	

0x56	pattern	
0x78	pattern	
0x9a	pattern	(no padding needed)
0x03	type	Type 3 - Upper-Layer Protocol
0x81	numeric_op	end-of-list, value size=1, ==
0x06	value	06

Table 2

This constitutes an NLRI with an NLRI length of 18 octets.

Padding is not needed either for the destination prefix pattern (length - offset = 32 bits) or for the source prefix pattern (length - offset = 40 bits), as both patterns end on an octet boundary.

3.8.2. Example 2

The following example demonstrates the prefix encoding for all packets from ::1234:5678:9a00:0/65-104 to 2001:db8::/32.

length	destination	source
0x0f	01 20 00 20 01 0d b8	02 68 41 24 68 ac f1 34

Table 3

Decoded:

Value		
0x0f	length	15 octets (if len<240, 1 octet)
0x01	type	Type 1 - Dest. IPv6 Prefix
0x20	length	32 bits
0x00	offset	0 bits
0x20	pattern	
0x01	pattern	
0x0d	pattern	
0xb8	pattern	(no padding needed)

0x02	type	Type 2 - Source IPv6 Prefix
0x68	length	104 bits
0x41	offset	65 bits
0x24	pattern	
0x68	pattern	
0xac	pattern	
0xf1	pattern	
0x34	pattern/pad	(contains 1 bit of padding)

Table 4

This constitutes an NLRI with an NLRI length of 15 octets.

The source prefix pattern is 104 - 65 = 39 bits in length. After the pattern, one bit of padding needs to be added so that the component ends on an octet boundary. However, only the first 39 bits are actually used for bitwise pattern matching, starting with a 65-bit offset from the topmost bit of the address.

4. Ordering of Flow Specifications

The definition for the order of traffic filtering rules from Section 5.1 of [RFC8955] is reused with new consideration for the IPv6 prefix offset. As long as the offsets are equal, the comparison is the same, retaining longest-prefix-match semantics. If the offsets are not equal, the lowest offset has precedence, as this Flow Specification matches the most significant bit.

The code in Appendix A shows a Python3 implementation of the resulting comparison algorithm. The full code was tested with Python 3.7.2 and can be obtained at <https://github.com/stoffi92/draft-ietf-idr-flow-spec-v6/tree/master/flowspec-cmp>.

5. Validation Procedure

The validation procedure is the same as specified in Section 6 of [RFC8955] with the exception that item a) of the validation procedure should now read as follows:

> a) A destination prefix component with offset=0 is embedded in the Flow Specification

6. IPv6 Traffic Filtering Action Changes

Traffic Filtering Actions from Section 7 of [RFC8955] can also be applied to IPv6 Flow Specifications. To allow an IPv6-Address-Specific Route-Target, a new Traffic Filtering Action IPv6-Address-Specific Extended Community is specified in Section 6.1 below.

6.1. Redirect IPv6 (rt-redirect-ipv6) Type 0x000d

The redirect IPv6-Address-Specific Extended Community allows the traffic to be redirected to a VRF routing instance that lists the specified IPv6-Address-Specific Route-Target in its import policy. If several local instances match this criteria, the choice between them is a local matter (for example, the instance with the lowest Route Distinguisher value can be elected).

This IPv6-Address-Specific Extended Community uses the same encoding as the IPv6-Address-Specific Route-Target Extended Community (Section 2 of [RFC5701]) with the Type value always 0x000d.

The Local Administrator subfield contains a number from a numbering space that is administered by the organization to which the IP address carried in the Global Administrator subfield has been assigned by an appropriate authority.

Interferes with: All BGP Flow Specification redirect Traffic Filtering Actions (with itself and those specified in Section 7.4 of [RFC8955]).

7. Security Considerations

This document extends the functionality in [RFC8955] to be applicable to IPv6 data packets. The same security considerations from [RFC8955] now also apply to IPv6 networks.

[RFC7112] describes the impact of oversized IPv6 header chains when trying to match on the transport header; Section 4.5 of [RFC8200] also requires that the first fragment must include the upper-layer header, but there could be wrongly formatted packets not respecting [RFC8200]. IPv6 Flow Specification component Type 3 (Section 3.3) will not be enforced for those illegal packets. Moreover, there are hardware limitations in several routers (Section 1 of [RFC8883]) that may make it impossible to enforce a policy signaled by a Type 3 Flow Specification component or Flow Specification components that match on upper-layer properties of the packet.

8. IANA Considerations

This section complies with [RFC7153].

8.1. Flow Spec IPv6 Component Types

IANA has created and maintains a registry entitled "Flow Spec Component Types". IANA has added this document as a reference for that registry. Furthermore, the registry has been updated to also contain the IPv6 Flow Specification Component Types as described below. The registration procedure remains unchanged.

8.1.1. Registry Template

Type Value: contains the assigned Flow Specification component type value

IPv4 Name: contains the associated IPv4 Flow Specification component name as specified in [RFC8955]

IPv6 Name: contains the associated IPv6 Flow Specification component name as specified in this document

Reference: contains references to the specifications

8.1.2. Registry Contents

Type Value: 0
IPv4 Name: Reserved
IPv6 Name: Reserved
Reference: [RFC8955], RFC 8956

Type Value: 1
IPv4 Name: Destination Prefix
IPv6 Name: Destination IPv6 Prefix
Reference: [RFC8955], RFC 8956

Type Value: 2
IPv4 Name: Source Prefix
IPv6 Name: Source IPv6 Prefix
Reference: [RFC8955], RFC 8956

Type Value: 3
IPv4 Name: IP Protocol
IPv6 Name: Upper-Layer Protocol
Reference: [RFC8955], RFC 8956

Type Value: 4
IPv4 Name: Port
IPv6 Name: Port
Reference: [RFC8955], RFC 8956

Type Value: 5
IPv4 Name: Destination Port
IPv6 Name: Destination Port
Reference: [RFC8955], RFC 8956

Type Value: 6
IPv4 Name: Source Port
IPv6 Name: Source Port
Reference: [RFC8955], RFC 8956

Type Value: 7
IPv4 Name: ICMP Type
IPv6 Name: ICMPv6 Type
Reference: [RFC8955], RFC 8956

Type Value: 8
IPv4 Name: ICMP Code
IPv6 Name: ICMPv6 Code
Reference: [RFC8955], RFC 8956

Type Value: 9
IPv4 Name: TCP Flags
IPv6 Name: TCP Flags
Reference: [RFC8955], RFC 8956

Type Value: 10
IPv4 Name: Packet Length
IPv6 Name: Packet Length
Reference: [RFC8955], RFC 8956

Type Value: 11
IPv4 Name: DSCP
IPv6 Name: DSCP
Reference: [RFC8955], RFC 8956

Type Value: 12
IPv4 Name: Fragment
IPv6 Name: Fragment
Reference: [RFC8955], RFC 8956

Type Value: 13
IPv4 Name: Unassigned
IPv6 Name: Flow Label
Reference: RFC 8956

Type Value: 14-254
IPv4 Name: Unassigned
IPv6 Name: Unassigned

Type Value: 255
IPv4 Name: Reserved
IPv6 Name: Reserved
Reference: [RFC8955], RFC 8956

8.2. IPv6-Address-Specific Extended Community Flow Spec IPv6 Actions

IANA maintains a registry entitled "Transitive IPv6-Address-Specific Extended Community Types". For the purpose of this work, IANA has assigned a new value:

Type Value	Name	Reference
0x000d	Flow spec rt-redirect-ipv6 format	RFC 8956

Table 5: Transitive IPv6-Address-Specific Extended Community Types Registry

9. Normative References

[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>.

[RFC4271] Rekhter, Y., Ed., Li, T., Ed., and S. Hares, Ed., "A Border Gateway Protocol 4 (BGP-4)", RFC 4271, DOI 10.17487/RFC4271, January 2006, <https://www.rfc-editor.org/info/rfc4271>.

[RFC4443] Conta, A., Deering, S., and M. Gupta, Ed., "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", STD 89, RFC 4443, DOI 10.17487/RFC4443, March 2006, <https://www.rfc-editor.org/info/rfc4443>.

[RFC4760] Bates, T., Chandra, R., Katz, D., and Y. Rekhter, "Multiprotocol Extensions for BGP-4", RFC 4760, DOI 10.17487/RFC4760, January 2007, <https://www.rfc-editor.org/info/rfc4760>.

[RFC5701] Rekhter, Y., "IPv6 Address Specific BGP Extended Community Attribute", RFC 5701, DOI 10.17487/RFC5701, November 2009, <https://www.rfc-editor.org/info/rfc5701>.

[RFC7112] Gont, F., Manral, V., and R. Bonica, "Implications of Oversized IPv6 Header Chains", RFC 7112, DOI 10.17487/RFC7112, January 2014, <https://www.rfc-editor.org/info/rfc7112>.

[RFC7153] Rosen, E. and Y. Rekhter, "IANA Registries for BGP Extended Communities", RFC 7153, DOI 10.17487/RFC7153, March 2014, <https://www.rfc-editor.org/info/rfc7153>.

[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>.

[RFC8200] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", STD 86, RFC 8200, DOI 10.17487/RFC8200, July 2017, <https://www.rfc-editor.org/info/rfc8200>.

[RFC8883] Herbert, T., "ICMPv6 Errors for Discarding Packets Due to Processing Limits", RFC 8883, DOI 10.17487/RFC8883, September 2020, <https://www.rfc-editor.org/info/rfc8883>.

[RFC8955] Loibl, C., Hares, S., Raszuk, R., McPherson, D., and M. Bacher, "Dissemination of Flow Specification Rules", RFC 8955, DOI 10.17487/RFC8955, December 2020, <https://www.rfc-editor.org/info/rfc8955>.

Appendix A. Example Python Code: flow_rule_cmp_v6

```
<CODE BEGINS>
"""
Copyright (c) 2020 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 Simplified 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).
"""

import itertools
import collections
import ipaddress


EQUAL = 0
A_HAS_PRECEDENCE = 1
B_HAS_PRECEDENCE = 2
IP_DESTINATION = 1
IP_SOURCE = 2

FS_component = collections.namedtuple('FS_component',
                                      'component_type value')


class FS_IPv6_prefix_component:
    def __init__(self, prefix, offset=0,
                 component_type=IP_DESTINATION):
        self.offset = offset
        self.component_type = component_type
        # make sure if offset != 0 that none of the
        # first offset bits are set in the prefix
        self.value = prefix
        if offset != 0:
            i = ipaddress.IPv6Interface(
                (self.value.network_address, offset))
            if i.network.network_address != \
                ipaddress.ip_address('0::0'):
                raise ValueError('Bits set in the offset')


class FS_nlri(object):
    """
    FS_nlri class implementation that allows sorting.

    By calling .sort() on an array of FS_nlri objects these
    will be sorted according to the flow_rule_cmp algorithm.

    Example:
    nlri = [ FS_nlri(components=[
             FS_component(component_type=4,
                          value=bytearray([0,1,2,3,4,5,6])),
             ]),
             FS_nlri(components=[
             FS_component(component_type=5,
                          value=bytearray([0,1,2,3,4,5,6])),
             FS_component(component_type=6,
                          value=bytearray([0,1,2,3,4,5,6])),
             ]),
           ]
    nlri.sort() # sorts the array according to the algorithm
```

```
    """
    def __init__(self, components = None):
        """
        components: list of type FS_component
        """
        self.components = components

    def __lt__(self, other):
        # use the below algorithm for sorting
        result = flow_rule_cmp_v6(self, other)
        if result == B_HAS_PRECEDENCE:
            return True
        else:
            return False


def flow_rule_cmp_v6(a, b):
    """
    Implementation of the flowspec sorting algorithm in
    RFC 8956.
    """
    for comp_a, comp_b in itertools.zip_longest(a.components,
                                                b.components):
        # If a component type does not exist in one rule
        # this rule has lower precedence
        if not comp_a:
            return B_HAS_PRECEDENCE
        if not comp_b:
            return A_HAS_PRECEDENCE
        # Higher precedence for lower component type
        if comp_a.component_type < comp_b.component_type:
            return A_HAS_PRECEDENCE
        if comp_a.component_type > comp_b.component_type:
            return B_HAS_PRECEDENCE
        # component types are equal -> type-specific comparison
        if comp_a.component_type in (IP_DESTINATION, IP_SOURCE):
            if comp_a.offset < comp_b.offset:
                return A_HAS_PRECEDENCE
            if comp_a.offset > comp_b.offset:
                return B_HAS_PRECEDENCE
            # both components have the same offset
            # assuming comp_a.value, comp_b.value of type
            # ipaddress.IPv6Network
            # and the offset bits are reset to 0 (since they are
            # not represented in the NLRI)
            if comp_a.value.overlaps(comp_b.value):
                # longest prefixlen has precedence
                if comp_a.value.prefixlen > \
                    comp_b.value.prefixlen:
                    return A_HAS_PRECEDENCE
                if comp_a.value.prefixlen < \
                    comp_b.value.prefixlen:
                    return B_HAS_PRECEDENCE
                # components equal -> continue with next
                # component
            elif comp_a.value > comp_b.value:
```

```
                    return B_HAS_PRECEDENCE
                elif comp_a.value < comp_b.value:
                    return A_HAS_PRECEDENCE
            else:
                # assuming comp_a.value, comp_b.value of type
                # bytearray
                if len(comp_a.value) == len(comp_b.value):
                    if comp_a.value > comp_b.value:
                        return B_HAS_PRECEDENCE
                    if comp_a.value < comp_b.value:
                        return A_HAS_PRECEDENCE
                    # components equal -> continue with next
                    # component
                else:
                    common = min(len(comp_a.value),
                                 len(comp_b.value))
                    if comp_a.value[:common] > \
                        comp_b.value[:common]:
                        return B_HAS_PRECEDENCE
                    elif comp_a.value[:common] < \
                          comp_b.value[:common]:
                        return A_HAS_PRECEDENCE
                    # the first common bytes match
                    elif len(comp_a.value) > len(comp_b.value):
                        return A_HAS_PRECEDENCE
                    else:
                        return B_HAS_PRECEDENCE
        return EQUAL
    <CODE ENDS>
```

Acknowledgments

The authors would like to thank Pedro Marques, Hannes Gredler, Bruno Rijsman, Brian Carpenter, and Thomas Mangin for their valuable input.

Contributors

Danny McPherson
Verisign, Inc.

Email: dmcpherson@verisign.com

Burjiz Pithawala
Individual

Email: burjizp@gmail.com

Andy Karch
Cisco Systems
170 West Tasman Drive
San Jose, CA 95134
United States of America

Email: akarch@cisco.com

Authors' Addresses

Christoph Loibl (editor)
next layer Telekom GmbH
Mariahilfer Guertel 37/7
1150 Vienna
Austria

Phone: +43 664 1176414
Email: cl@tix.at

Robert Raszuk (editor)
NTT Network Innovations
940 Stewart Dr
Sunnyvale, CA 94085
United States of America

Email: robert@raszuk.net

Susan Hares (editor)
Huawei
7453 Hickory Hill
Saline, MI 48176
United States of America

Email: shares@ndzh.com