Network Working Group
Request for Comments: 5225
Category: Standards Track

G. Pelletier
K. Sandlund
Ericsson

April 2008

RObust Header Compression Version 2 (ROHCv2): Profiles for RTP, UDP, IP, ESP and UDP-Lite

Status of This Memo

Abstract

This document specifies ROHC (Robust Header Compression) profiles that efficiently compress RTP/UDP/IP (Real-Time Transport Protocol, User Datagram Protocol, Internet Protocol), RTP/UDP-Lite/IP (User Datagram Protocol Lite), UDP/IP, UDP-Lite/IP, IP and ESP/IP (Encapsulating Security Payload) headers.

This specification defines a second version of the profiles found in RFC 3095, RFC 3843 and RFC 4019; it supersedes their definition, but does not obsolete them.

The ROHCv2 profiles introduce a number of simplifications to the rules and algorithms that govern the behavior of the compression endpoints. It also defines robustness mechanisms that may be used by a compressor implementation to increase the probability of decompression success when packets can be lost and/or reordered on the ROHC channel. Finally, the ROHCv2 profiles define their own specific set of header formats, using the ROHC formal notation.

Table of Contents

1. Introduction

The ROHC WG has developed a header compression framework on top of which various profiles can be defined for different protocol sets or compression requirements. The ROHC framework was first documented in [RFC3095], together with profiles for compression of RTP/UDP/IP (Real-Time Transport Protocol, User Datagram Protocol, Internet Protocol), UDP/IP, IP and ESP/IP (Encapsulating Security Payload) headers. Additional profiles for compression of IP headers [RFC3843] and UDP-Lite (User Datagram Protocol Lite) headers [RFC4019] were later specified to complete the initial set of ROHC profiles.

This document defines an updated version for each of the above mentioned profiles, and the definitions depend on the ROHC framework as found in [RFC4995]. The framework is required reading to understand the profile definitions, rules, and their role.

Specifically, this document defines header compression schemes for:

o RTP/UDP/IP : profile 0x0101
o UDP/IP : profile 0x0102
o ESP/IP : profile 0x0103
o IP : profile 0x0104
o RTP/UDP-Lite/IP : profile 0x0107
o UDP-Lite/IP : profile 0x0108

Each of the profiles above can compress the following type of extension headers:

- AH [RFC4302]
- GRE [RFC2784][RFC2890]
- MINE [RFC2004]
- IPv6 Destination Options header[RFC2460]
- IPv6 Hop-by-hop Options header[RFC2460]
- IPv6 Routing header [RFC2460]

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

This document is consistent with the terminology found in the ROHC framework [RFC4995] and in the formal notation for ROHC [RFC4997]. In addition, this document uses or defines the following terms:

Acknowledgment Number

The Acknowledgment Number identifies what packet is being acknowledged in the RoHCv2 feedback element (See Section 6.9). The value of this field normally corresponds to the Master Sequence Number (MSN) of the header that was last successfully decompressed, for the compression context (CID) for which the feedback information applies.

Chaining of Items

A chain of items groups fields based on similar characteristics. ROHCv2 defines chain items for static, dynamic and irregular fields. Chaining is achieved by appending an item to the chain for each header in its order of appearance in the uncompressed packet. Chaining is useful to construct compressed headers from an arbitrary number of any of the protocol headers for which a ROHCv2 profile defines a compressed format.

CRC-3 Control Fields Validation

The CRC-3 control fields validation refers to the validation of the control fields. This validation is performed by the decompressor when it receives a Compressed (CO) header that contains a 3-bit Cyclic Redundancy Check (CRC) calculated over control fields. This 3-bit CRC covers controls fields carried in the CO header as well as specific control fields in the context. In the formal definition of the header formats, this 3-bit CRC is labeled "control_crc3" and uses the control_crc3_encoding (See also Section 6.6.11).

Delta

The delta refers to the difference in the absolute value of a field between two consecutive packets being processed by the same compression endpoint.

Reordering Depth

The number of packets by which a packet is received late within its sequence due to reordering between the compressor and the decompressor, i.e., reordering between packets associated with the same context (CID). See the definition of sequentially late packet below.

ROHCv2 Header Types

ROHCv2 profiles use two different header types: the Initialization and Refresh (IR) header type, and the Compressed (CO) header type.

Sequentially Early Packet

A packet that reaches the decompressor before one or several packets that were delayed over the channel, where all of the said packets belong to the same header-compressed flow and are associated to the same compression context (CID). At the time of the arrival of a sequentially early packet, the packet(s) delayed on the link cannot be differentiated from lost packet(s).

Sequentially Late Packet

A packet is late within its sequence if it reaches the decompressor after one or several other packets belonging to the same CID have been received, although the sequentially late packet was sent from the compressor before the other packet(s). How the decompressor detects a sequentially late packet is outside the scope of this specification, but it can for example use the MSN for this purpose.

Timestamp Stride (ts_stride)

The timestamp stride (ts_stride) is the expected increase in the timestamp value between two RTP packets with consecutive sequence numbers. For example, for a media encoding with a sample rate of 8 kHz producing one frame every 20 ms, the RTP timestamp will typically increase by n * 160 (= 8000 * 0.02), for some integer n.

Time Stride (time_stride)

The time stride (time_stride) is the time interval equivalent to one ts_stride, e.g., 20 ms in the example for the RTP timestamp increment above.

3. Acronyms

This section lists most acronyms used for reference, in addition to those defined in [RFC4995].

AH	Authentication Header.
ESP	Encapsulating Security Payload.
GRE	Generic Routing Encapsulation.
FC	Full Context state (decompressor).
IP	Internet Protocol.
LSB	Least Significant Bits.
MINE	Minimal Encapsulation in IP.
MSB	Most Significant Bits.
MSN	Master Sequence Number.
NC	No Context state (decompressor).
OA	Optimistic Approach.
RC	Repair Context state (decompressor).
ROHC	Header compression framework (RFC 4995).
ROHCv2	Set of header compression profiles defined in this document.
RTP	Real-time Transport Protocol.
SSRC	Synchronization source. Field in RTP header.
CSRC	Contributing source. The RTP header contains an optional list of contributing sources.
TC	Traffic Class. Field in the IPv6 header. See also TOS.
TOS	Type Of Service. Field in the IPv4 header. See also TC.
TS	RTP Timestamp.
TTL	Time to Live. Field in the IPv4 header.
UDP	User Datagram Protocol.
UDP-Lite	User Datagram Protocol Lite.

4. Background (Informative)

This section provides background information on the compression profiles defined in this document. The fundamentals of general header compression and of the ROHC framework may be found in sections 3 and 4 of [RFC4995], respectively. The fundamentals of the formal notation for ROHC are defined in [RFC4997]. [RFC4224] describes the impacts of out-of-order delivery on profiles based on [RFC3095].

4.1. Classification of Header Fields

Section 3.1 of [RFC4995] explains that header compression is possible due to the fact that there is much redundancy between field values within the headers of a packet, especially between the headers of consecutive packets.

Appendix A lists and classifies in detail all the header fields relevant to this document. The appendix concludes with

recommendations on how the various fields should be handled by header compression algorithms.

The main conclusion is that most of the header fields can easily be compressed away since they never or seldom change. A small number of fields however need more sophisticated mechanisms.

These fields are:

- IPv4 Identification (16 bits) - IP-ID
- ESP Sequence Number (32 bits) - ESP SN
- UDP Checksum (16 bits) - Checksum
- UDP-Lite Checksum (16 bits) - Checksum
- UDP-Lite Checksum Coverage (16 bits) - CCov
- RTP Marker (1 bit) - M-bit
- RTP Sequence Number (16 bits) - RTP SN
- RTP Timestamp (32 bits) - TS

In particular, for RTP, the analysis in Appendix A reveals that the values of the RTP Timestamp (TS) field usually have a strong correlation to the RTP Sequence Number (SN), which increments by one for each packet emitted by an RTP source. The RTP M-bit is expected to have the same value most of the time, but it needs to be communicated explicitly on occasion.

For UDP, the Checksum field cannot be inferred or recalculated at the receiving end without violating its end-to-end properties, and is thus sent as-is when enabled (mandatory with IPv6). The same applies to the UDP-Lite Checksum (mandatory with both IPv4 and IPv6), while the UDP-Lite Checksum Coverage may in some cases be compressible.

For IPv4, a similar correlation as that of the RTP TS to the RTP SN is often observed between the Identifier field (IP-ID) and the master sequence number (MSN) used for compression (e.g., the RTP SN when compressing RTP headers).

4.2. Improvements of ROHCv2 over RFC 3095 Profiles

The ROHCv2 profiles can achieve compression efficiency and robustness that are both at least equivalent to RFC 3095 profiles [RFC3095], when used under the same operating conditions. In particular, the size and bit layout of the smallest compressed header (i.e., PT-0 format U/O-0 in RFC 3095, and pt_0_crc3 in ROHCv2) are identical.

There are a number of differences and improvements between profiles defined in this document and their earlier version defined in RFC 3095. This section provides an overview of some of the most significant improvements:

Tolerance to reordering

Profiles defined in RFC 3095 require that the channel between compressor and decompressor provide in-order delivery between compression endpoints. ROHCv2 profiles, however, can handle robustly and efficiently a limited amount of reordering after the compression point as part of the compression algorithm itself. In addition, this improved support for reordering makes it possible for ROHCv2 profiles to handle prelink reordering more efficiently.

Operational logic

Profiles in RFC 3095 define multiple operational modes, each with different updating logic and compressed header formats. ROHCv2 profiles operate in unidirectional operation until feedback is first received for a context (CID), at which point bidirectional operation is used; the formats are independent of what operational logic is used.

IP extension header

Profiles in RFC 3095 compress IP Extension headers using list compression. ROHCv2 profiles instead treat extension headers in the same manner as other protocol headers, i.e., using the chaining mechanism; it thus assumes that extension headers are not added or removed during the lifetime of a context (CID), otherwise compression has to be restarted for this flow.

IP encapsulation

Profiles in RFC 3095 can compress at most two levels of IP headers. ROHCv2 profiles can compress an arbitrary number of IP headers.

List compression

ROHCv2 profiles do not support reference-based list compression.

Robustness and repairs

ROHCv2 profiles do not define a format for the IR-DYN packet; instead, each profile defines a compressed header that can be used to perform a more robust context repair using a 7-bit CRC verification. This also implies that only the IR header can change the association between a CID and the profile it uses.

Feedback

ROHCv2 profiles mandate a CRC in the format of the FEEDBACK-2, while this is optional in RFC 3095. A different set of feedback options is also used in ROHCv2 compared to RFC 3095.

4.3. Operational Characteristics of ROHCv2 Profiles

Robust header compression can be used over different link technologies. Section 4.4 of [RFC4995] lists the operational characteristics of the ROHC channel. The ROHCv2 profiles address a wide range of applications, and this section summarizes some of the operational characteristics that are specific to these profiles.

Packet length

ROHCv2 profiles assume that the lower layer indicates the length of a compressed packet. ROHCv2 compressed headers do not contain length information for the payload.

Out-of-order delivery between compression endpoints

The definition of the ROHCv2 profiles places no strict requirement on the delivery sequence between the compression endpoints, i.e., packets may be received in a different order than the compressor has sent them and still have a fair probability of being successfully decompressed.

However, frequent out-of-order delivery and/or significant reordering depth will negatively impact the compression efficiency. More specifically, if the compressor can operate using a proper estimate of the reordering characteristics of the path between the compression endpoints, larger headers can be sent more often to increase the robustness against decompression failures due to out-of-order delivery. Otherwise, the compression efficiency will be impaired from an increase in the frequency of decompression failures and recovery attempts.

5. Overview of the ROHCv2 Profiles (Informative)

This section provides an overview of concepts that are important and useful to the ROHCv2 profiles. These concepts may be used as guidelines for implementations but they are not part of the normative definition of the profiles, as these concepts relate to the compression efficiency of the protocol without impacting the interoperability characteristics of an implementation.

5.1. Compressor Concepts

Header compression can be conceptually characterized as the interaction of a compressor with a decompressor state machine, one per context. The responsibility of the compressor is to convey the information needed to successfully decompress a packet, based on a certain confidence regarding the state of the decompressor context.

This confidence is obtained from the frequency and the type of information the compressor sends when updating the decompressor context from the optimistic approach (Section 5.1.1), and optionally from feedback messages (See Section 6.9), received from the decompressor.

5.1.1. Optimistic Approach

A compressor always uses the optimistic approach when it performs context updates. The compressor normally repeats the same type of update until it is fairly confident that the decompressor has successfully received the information. If the decompressor successfully receives any of the headers containing this update, the state will be available for the decompressor to process smaller compressed headers.

If field X in the uncompressed header changes value, the compressor uses a header type that contains an encoding of field X until it has gained confidence that the decompressor has received at least one packet containing the new value for X. The compressor normally selects a compressed format with the smallest header that can convey the changes needed to achieve confidence.

The number of repetitions that is needed to obtain this confidence is normally related to the packet loss and out-of-order delivery characteristics of the link where header compression is used; it is thus not defined in this document. It is outside the scope of this specification and is left to implementors to decide.

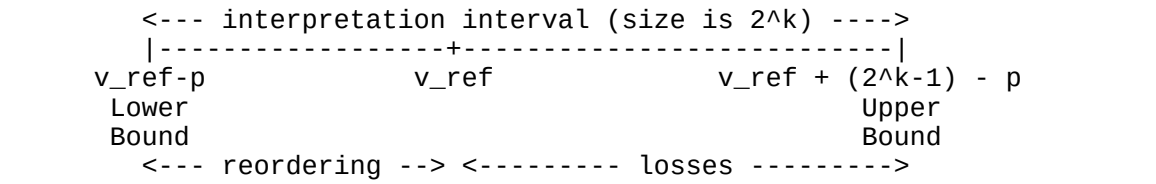
5.1.2. Tradeoff between Robustness to Losses and to Reordering

The ability of a header compression algorithm to handle sequentially late packets is mainly limited by two factors: the interpretation interval offset of the sliding window used for lsb encoded fields [RFC4997], and the optimistic approach (See Section 5.1.1) for seldom changing fields.

lsb encoded fields:

The interpretation interval offset specifies an upper limit for the maximum reordering depth, by which is it possible for the decompressor to recover the original value of a dynamically changing (i.e., sequentially incrementing) field that is encoded using a window-based lsb encoding. Its value is typically bound to the number of lsb compressed bits in the compressed header format, and thus grows with the number of bits transmitted. However, the offset and the lsb encoding only provide robustness for the field that it compresses, and (implicitly) for other sequentially changing fields that are derived from that field.

This is shown in the figure below:



where p is the maximum negative delta, corresponding to the maximum reordering depth for which the lsb encoding can recover the original value of the field;

where (2^k-1) - p is the maximum positive delta, corresponding to the maximum number of consecutive losses for which the lsb encoding can recover the original value of the field;

where v_ref is the reference value, as defined in the lsb encoding method in [RFC4997].

There is thus a tradeoff between the robustness against reordering and the robustness against packet losses, with respect to the number of MSN bits needed and the distribution of the interpretation interval between negative and positive deltas in the MSN.

Seldom changing fields

The optimistic approach (Section 5.1.1) provides the upper limit for the maximum reordering depth for seldom changing fields.

There is thus a tradeoff between compression efficiency and robustness. When only information on the MSN needs to be conveyed to the decompressor, the tradeoff relates to the number of compressed

MSN bits in the compressed header format. Otherwise, the tradeoff relates to the implementation of the optimistic approach.

In particular, compressor implementations should adjust their optimistic approach strategy to match both packet loss and reordering characteristics of the link over which header compression is applied. For example, the number of repetitions for each update of a non-lsb encoded field can be increased. The compressor can ensure that each update is repeated until it is reasonably confident that at least one packet containing the change has reached the decompressor before the first packet sent after this sequence.

5.1.3. Interactions with the Decompressor Context

The compressor normally starts compression with the initial assumption that the decompressor has no useful information to process the new flow, and sends Initialization and Refresh (IR) packets.

Initially, when sending the first IR packet for a compressed flow, the compressor does not expect to receive feedback for that flow, until such feedback is first received. At this point, the compressor may then assume that the decompressor will continue to send feedback in order to repair its context when necessary. The former is referred to as unidirectional operation, while the latter is called bidirectional operation.

The compressor can then adjust the compression level (i.e., what header format it selects) based on its confidence that the decompressor has the necessary information to successfully process the compressed headers that it selects.

In other words, the responsibilities of the compressor are to ensure that the decompressor operates with state information that is sufficient to successfully decompress the type of compressed header(s) it receives, and to allow the decompressor to successfully recover that state information as soon as possible otherwise. The compressor therefore selects the type of compressed header based on the following factors:

- the outcome of the encoding method applied to each field;
- the optimistic approach, with respect to the characteristics of the channel;
- the type of operation (unidirectional or bidirectional), and if in bidirectional operation, feedback received from the decompressor (ACKs, NACKs, STATIC-NACK, and options).

Encoding methods normally use previous value(s) from a history of packets whose headers it has previously compressed. The optimistic approach is meant to ensure that at least one compressed header containing the information to update the state for a field is received. Finally, feedback indicates what actions the decompressor has taken with respect to its assumptions regarding the validity of its context (Section 5.2.2); it indicates what type of compressed header the decompressor can or cannot decompress.

The decompressor has the means to detect decompression failures for any compressed (CO) header format, using the CRC verification. Depending on the frequency and/or on the type of the failure, it might send a negative acknowledgement (NACK) or an explicit request for a complete context update (STATIC-NACK). However, the decompressor does not have the means to identify the cause of the failure, and in particular the decompression of what field(s) is responsible for the failure. The compressor is thus always responsible for determining the most suitable response to a negative acknowledgement, using the confidence it has in the state of the decompressor context, when selecting the type of compressed header it will use when compressing a header.

5.2. Decompressor Concepts

The decompressor normally uses the last received and successfully validated (IR packets) or verified (CO packets) header as the reference for future decompression.

The decompressor is responsible for verifying the outcome of every decompression attempt, to update its context when successful, and finally to request context repairs by making coherent usage of feedback once it has started using feedback.

Specifically, the outcome of every decompression attempt is verified using the CRC present in the compressed header; the decompressor updates the context information when this outcome is successfully verified; finally, if the decompressor uses feedback once for a compressed flow, then it will continue to do so for as long as the corresponding context is associated with the same profile.

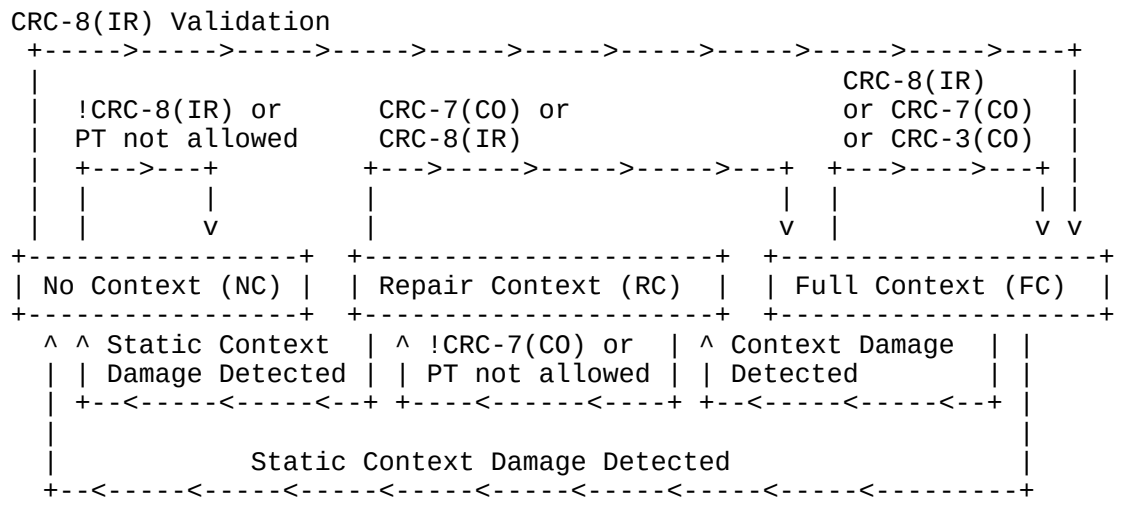
5.2.1. Decompressor State Machine

The decompressor operation may be represented as a state machine defining three states: No Context (NC), Repair Context (RC), and Full Context (FC).

The decompressor starts without a valid context, the NC state. Upon receiving an IR packet, the decompressor validates the integrity of

its header using the CRC-8 validation. If the IR header is successfully validated, the decompressor updates the context and uses this header as the reference header, and moves to the FC state. Once the decompressor state machine has entered the FC state, it does not normally leave; only repeated decompression failures will force the decompressor to transit downwards to a lower state. When context damage is detected, the decompressor moves to the repair context (RC) state, where it stays until it successfully verifies a decompression attempt for a compressed header with a 7-bit CRC or until it successfully validates an IR header. When static context damage is detected, the decompressor moves back to the NC state.

Below is the state machine for the decompressor. Details of the transitions between states and decompression logic are given in the sub-sections following the figure.



where:

CRC-8(IR)	: Successful CRC-8 validation for the IR header.
!CRC-8(IR)	: Unsuccessful CRC-8 validation for the IR header.
CRC-7(CO) and/or CRC-3(CO)	: Successful CRC verification for the decompression of a CO header, based on the number of CRC bits carried in the CO header.
!CRC-7(CO)	: Failure to CRC verify the decompression of a CO header carrying a 7-bit CRC.
PT not allowed	: The decompressor has received a packet type (PT) for which the decompressor's current context does not provide enough valid state information to decompress the packet.

Static Context Damage Detected: See definition in Section 5.2.2.

Context Damage Detected: See definition in Section 5.2.2.

5.2.1.1. No Context (NC) State

Initially, while working in the No Context (NC) state, the decompressor has not yet successfully validated an IR header.

Attempting decompression:

In the NC state, only packets carrying sufficient information on the static fields (i.e., IR packets) can be decompressed.

Upward transition:

The decompressor can move to the Full Context (FC) state when the CRC validation of an 8-bit CRC in an IR header is successful.

Feedback logic:

In the NC state, the decompressor should send a STATIC-NACK if a packet of a type other than IR is received, or if an IR header has failed the CRC-8 validation, subject to the feedback rate limitation as described in Section 5.2.3.

5.2.1.2. Repair Context (RC) State

In the Repair Context (RC) state, the decompressor has successfully decompressed packets for this context, but does not have confidence that the entire context is valid.

Attempting decompression:

In the RC state, only headers covered by an 8-bit CRC (i.e., IR) or CO headers carrying a 7-bit CRC can be decompressed.

Upward transition:

The decompressor can move to the Full Context (FC) state when the CRC verification succeeds for a CO header carrying a 7-bit CRC or when validation of an 8-bit CRC in an IR header succeeds.

Downward transition:

The decompressor moves back to the NC state if it assumes static context damage.

Feedback logic:

In the RC state, the decompressor should send a STATIC-NACK when CRC-8 validation of an IR header fails, or when a CO header carrying a 7-bit CRC fails and static context damage is assumed, subject to the feedback rate limitation as described in Section 5.2.3. If any other packet type is received, the decompressor should treat it as a CRC verification failure to determine if NACK is to be sent.

5.2.1.3. Full Context (FC) State

In the Full Context (FC) state, the decompressor assumes that its entire context is valid.

Attempting decompression:

In the FC state, decompression can be attempted regardless of the type of packet received.

Downward transition:

The decompressor moves back to the RC state if it assumes context damage. If the decompressor assumes static context damage, it moves directly to the NC state.

Feedback logic:

In the FC state, the decompressor should send a NACK when CRC-8 validation or CRC verification of any header type fails and if context damage is assumed, or it should send a STATIC-NACK if static context damage is assumed; this is subject to the feedback rate limitation described in Section 5.2.3.

5.2.2. Decompressor Context Management

All header formats carry a CRC and are context updating. A packet for which the CRC succeeds updates the reference values of all header fields, either explicitly (from the information about a field carried within the compressed header) or implicitly (fields inferred from other fields).

The decompressor may assume that some or the entire context is invalid, when it fails to validate or to verify one or more headers using the CRC. Because the decompressor cannot know the exact

reason(s) for a CRC failure or what field caused it, the validity of the context hence does not refer to what specific part(s) of the context is deemed valid or not.

Validity of the context rather relates to the detection of a problem with the context. The decompressor first assumes that the type of information that most likely caused the failure(s) is the state that normally changes for each packet, i.e., context damage of the dynamic part of the context. Upon repeated decompression failures and unsuccessful repairs, the decompressor then assumes that the entire context, including the static part, needs to be repaired, i.e., static context damage. Failure to validate the 3-bit CRC that protects control fields should be treated as a decompression failure when the decompressor asserts the validity of its context.

Context Damage Detection

The assumption of context damage means that the decompressor will not attempt decompression of a CO header that carries only a 3-bit CRC, and will only attempt decompression of IR headers or CO headers protected by a CRC-7.

Static Context Damage Detection

The assumption of static context damage means that the decompressor refrains from attempting decompression of any type of header other than the IR header.

How these assumptions are made, i.e., how context damage is detected, is open to implementations. It can be based on the residual error rate, where a low error rate makes the decompressor assume damage more often than on a high rate link.

The decompressor implements these assumptions by selecting the type of compressed header for which it will attempt decompression. In other words, validity of the context refers to the ability of a decompressor to attempt (or not) decompression of specific packet types.

When ROHCv2 profiles are used over a channel that cannot guarantee in-order delivery, the decompressor may refrain from updating its context with the content of a sequentially late packet that is successfully decompressed. This is to avoid updating the context with information that is older than what the decompressor already has in its context.

5.2.3. Feedback Logic

ROHCv2 profiles may be used in environments with or without feedback capabilities from decompressor to compressor. ROHCv2 however assumes that if a ROHC feedback channel is available and if this channel is used at least once by the decompressor for a specific context, this channel will be used during the entire compression operation for that context (i.e., bidirectional operation).

The ROHC framework defines 3 types of feedback messages: ACKs, NACKs, and STATIC-NACKs. The semantics of each message is defined in Section 5.2.4.1. of [RFC4995]. What feedback to send is coupled with the context management of the decompressor, i.e., with the implementation of the context damage detection algorithms as described in Section 5.2.2.

The decompressor should send a NACK when it assumes context damage, and it should send a STATIC-NACK when it assumes static context damage. The decompressor is not strictly expected to send ACK feedback upon successful decompression, other than for the purpose of improving compression efficiency.

When ROHCv2 profiles are used over a channel that cannot guarantee in-order delivery, the decompressor may refrain from sending ACK feedback for a sequentially late packet that is successfully decompressed.

The decompressor should limit the rate at which it sends feedback, for both ACKs and STATIC-NACK/NACKs, and should avoid sending unnecessary duplicates of the same type of feedback message that may be associated with the same event.

6. ROHCv2 Profiles (Normative)

6.1. Channel Parameters, Segmentation, and Reordering

The compressor MUST NOT use ROHC segmentation (see Section 5.2.5 of [RFC4995]), i.e., the Maximum Reconstructed Reception Unit (MRRU) MUST be set to 0, if the configuration of the ROHC channel contains at least one ROHCv2 profile in the list of supported profiles (i.e., the PROFILES parameter) and if the channel cannot guarantee in-order delivery of packets between compression endpoints.

6.2. Profile Operation, Per-context

ROHCv2 profiles operate differently, per context, depending on how the decompressor makes use of the feedback channel, if any. Once the decompressor uses the feedback channel for a context, it establishes the feedback channel for that CID.

The compressor always starts with the assumption that the decompressor will not send feedback when it initializes a new context (see also the definition of a new context in Section 5.1.1. of [RFC4995], i.e., there is no established feedback channel for the new context. At this point, despite the use of the optimistic approach, decompression failure is still possible because the decompressor may not have received sufficient information to correctly decompress the packets; therefore, until the decompressor has established a feedback channel, the compressor SHOULD periodically send IR packets. The periodicity can be based on timeouts, on the number of compressed packets sent for the flow, or any other strategy the implementer chooses.

The reception of either positive feedback (ACKs) or negative feedback (NACKs or STATIC-NACKs) from the decompressor establishes the feedback channel for the context (CID) for which the feedback was received. Once there is an established feedback channel for a specific context, the compressor can make use of this feedback to estimate the current state of the decompressor. This helps to increase the compression efficiency by providing the information needed for the compressor to achieve the necessary confidence level. When the feedback channel is established, it becomes superfluous for the compressor to send periodic refreshes, and instead it can rely entirely on the optimistic approach and feedback from the decompressor.

The decompressor MAY send positive feedback (ACKs) to initially establish the feedback channel for a particular flow. Either positive feedback (ACKs) or negative feedback (NACKs or STATIC-NACKs) establishes this channel. Once it has established a feedback channel for a CID, the decompressor is REQUIRED to continue sending feedback for the lifetime of the context (i.e., until it receives an IR packet that associates the CID to a different profile), to send error recovery requests and (optionally) acknowledgments of significant context updates.

Compression without an established feedback channel will be less efficient, because of the periodic refreshes and the lack of feedback to trigger error recovery; there will also be a slightly higher probability of loss propagation compared to the case where the decompressor uses feedback.

6.3. Control Fields

ROHCv2 defines a number of control fields that are used by the decompressor in its interpretation of the header formats received from the compressor. The control fields listed in the following subsections are defined using the formal notation [RFC4997] in Section 6.8.2.4 of this document.

6.3.1. Master Sequence Number (MSN)

The Master Sequence Number (MSN) field is either taken from a field that already exists in one of the headers of the protocol that the profile compresses (e.g., RTP SN), or alternatively it is created at the compressor. There is one MSN space per context.

The MSN field has the following two functions:

- Differentiating between reference headers when receiving feedback data;
- Inferring the value of incrementing fields (e.g., IPv4 Identifier).

There is one MSN field in every ROHCv2 header, i.e., the MSN is always present in each header type sent by the compressor. The MSN is sent in full in IR headers, while it can be lsb encoded within CO header formats. The decompressor always includes LSBs of the MSN in the Acknowledgment Number field in feedback (see Section 6.9). The compressor can later use this field to infer what packet the decompressor is acknowledging.

For profiles for which the MSN is created by the compressor (i.e., 0x0102, 0x0104, and 0x0108), the following applies:

- The compressor only initializes the MSN for a context when that context is first created or when the profile associated with a context changes;
- When the MSN is initialized, it is initialized to a random value;
- The value of the MSN SHOULD be incremented by one for each packet that the compressor sends for a specific CID.

6.3.2. Reordering Ratio

The control field reorder_ratio specifies how much reordering is handled by the lsb encoding of the MSN. This is useful when header compression is performed over links with varying reordering

characteristics. The reorder_ratio control field provides the means for the compressor to adjust the robustness characteristics of the lsb encoding method with respect to reordering and consecutive losses, as described in Section 5.1.2.

6.3.3. IP-ID Behavior

The IP-ID field of the IPv4 header can have different change patterns: sequential in network byte order, sequential byte-swapped, random or constant (a constant value of zero, although not conformant with [RFC0791], has been observed in practice). There is one IP-ID behavior control field per IP header. The control field for the IP-ID behavior of the innermost IP header determines which set of header formats is used. The IP-ID behavior control field is also used to determine the contents of the irregular chain item, for each IP header.

ROHCv2 profiles MUST NOT assign a sequential behavior (network byte order or byte-swapped) to any IP-ID but the one in the innermost IP header when compressing more than one level of IP headers. This is because only the IP-ID of the innermost IP header is likely to have a sufficiently close correlation with the MSN to compress it as a sequentially changing field. Therefore, a compressor MUST assign either the constant zero IP-ID or the random IP-ID behavior to tunneling headers.

6.3.4. UDP-Lite Coverage Behavior

The control field coverage_behavior specifies how the checksum coverage field of the UDP-Lite header is compressed with RoHCv2. It can indicate one of the following encoding methods: irregular, static, or inferred encoding.

6.3.5. Timestamp Stride

The ts_stride control field is used in scaled RTP timestamp encoding (see Section 6.6.8). It defines the expected increase in the RTP timestamp between consecutive RTP sequence numbers.

6.3.6. Time Stride

The time_stride control field is used in timer-based compression encoding (see Section 6.6.9). When timer-based compression is used, time_stride should be set to the expected difference in arrival time between consecutive RTP packets.

6.3.7. CRC-3 for Control Fields

ROHCv2 profiles define a CRC-3 calculated over a number of control fields. This 3-bit CRC protecting the control fields is present in the header format for the co_common and co_repair header types.

The decompressor MUST always validate the integrity of the control fields covered by this 3-bit CRC when processing a co_common or a co_repair compressed header.

Failure to validate the control fields using this CRC should be considered as a decompression failure by the decompressor in the algorithm that assesses the validity of the context. However, if the decompression attempt can be verified using either the CRC-3 or the CRC-7 calculated over the uncompressed header, the decompressor MAY still forward the decompressed header to upper layers. This is because the protected control fields are not always used to decompress the header (i.e., co_common or co_repair) that updates their respective value.

The CRC polynomial and coverage of this CRC-3 is defined in Section 6.6.11.

6.4. Reconstruction and Verification

Validation of the IR header (8-bit CRC)

The decompressor MUST always validate the integrity of the IR header using the 8-bit CRC carried within the IR header. When the header is validated, the decompressor updates the context with the information in the IR header. Otherwise, if the IR cannot be validated, the context MUST NOT be updated and the IR header MUST NOT be delivered to upper layers.

Verification of CO headers (3-bit CRC or 7-bit CRC)

The decompressor MUST always verify the decompression of a CO header using the CRC carried within the compressed header. When the decompression is verified and successful, the decompressor updates the context with the information received in the CO header; otherwise, if the reconstructed header fails the CRC verification, these updates MUST NOT be performed.

A packet for which the decompression attempt cannot be verified using the CRC MUST NOT be delivered to upper layers.

Decompressor implementations may attempt corrective or repair measures on CO headers prior to performing the above actions, and the result of any decompression attempt MUST be verified using the CRC.

6.5. Compressed Header Chains

Some header types use one or more chains containing sub-header information. The function of a chain is to group fields based on similar characteristics, such as static, dynamic, or irregular fields.

Chaining is done by appending an item for each header to the chain in their order of appearance in the uncompressed packet, starting from the fields in the outermost header.

In the text below, the term <protocol_name> is used to identify formal notation names corresponding to different protocol headers. The mapping between these is defined in the following table:

Protocol		protocol_name
IPv4	RFC 0791	ipv4
IPv6	RFC 2460	ipv6
UDP	RFC 0768	udp
RTP	RFC 3550	rtp
ESP	RFC 4303	esp
UDP-Lite	RFC 3828	udp_lite
AH	RFC 4302	ah
GRE	RFC 2784, RFC 2890	gre
MINE	RFC 2004	mine
IPv6 Destination Option	RFC 2460	dest_opt
IPv6 Hop-by-hop Options	RFC 2460	hop_opt
IPv6 Routing Header	RFC 2460	rout_opt

Static chain:

The static chain consists of one item for each header of the chain of protocol headers that is compressed, starting from the outermost IP header. In the formal description of the header formats, this static chain item for each header type is labeled <protocol_name>_static. The static chain is only used in the IR header format.

Dynamic chain:

The dynamic chain consists of one item for each header of the chain of protocol headers that is compressed, starting from the outermost IP header. In the formal description of the header formats, the dynamic chain item for each header type is labeled <protocol_name>_dynamic. The dynamic chain is only used in the IR and co_repair header formats.

Irregular chain:

The structure of the irregular chain is analogous to the structure of the static chain. For each compressed header that uses the general format of Section 6.8, the irregular chain is appended at a specific location in the general format of the compressed headers. In the formal description of the header formats, the irregular chain item for each header type is a format whose name is suffixed by "_irregular". The irregular chain is used in all CO headers, except for the co_repair format.

The format of the irregular chain for the innermost IP header differs from the format used for the outer IP headers, because the innermost IP header is part of the compressed base header. In the definition of the header formats using the formal notation, the argument "is_innermost", which is passed to the corresponding encoding method (ipv4 or ipv6), determines what irregular chain items to use. The format of the irregular chain item for the outer IP headers is also determined using one flag for TTL/Hop Limit and TOS/TC. This flag is defined in the format of some of the compressed base headers.

ROHCv2 profiles compress extension headers as other headers, and thus extension headers have a static chain, a dynamic chain, and an irregular chain.

ROHCv2 profiles define chains for all headers that can be compressed, i.e., RTP [RFC3550], UDP [RFC0768], ESP [RFC4303], UDP-Lite [RFC3828], IPv4 [RFC0791], IPv6 [RFC2460], AH [RFC4302], GRE [RFC2784][RFC2890], MINE [RFC2004], IPv6 Destination Options header [RFC2460], IPv6 Hop-by-hop Options header [RFC2460], and IPv6 Routing header [RFC2460].

6.6. Header Formats and Encoding Methods

The header formats are defined using the ROHC formal notation. Some of the encoding methods used in the header formats are defined in [RFC4997], while other methods are defined in this section.

6.6.1. baseheader_extension_headers

The baseheader_extension_headers encoding method skips over all fields of the extension headers of the innermost IP header, without encoding any of them. Fields in these extension headers are instead encoded in the irregular chain.

This encoding is used in CO headers (see Section 6.8.2). The innermost IP header is combined with other header(s) (i.e., UDP, UDP-Lite, RTP) to create the compressed base header. In this case, there may be a number of extension headers between the IP headers and the other headers.

The base header defines a representation of the extension headers, to comply with the syntax of the formal notation; this encoding method provides this representation.

6.6.2. baseheader_outer_headers

The baseheader_outer_headers encoding method skips over all the fields of the extension header(s) that do not belong to the innermost IP header, without encoding any of them. Changing fields in outer headers are instead handled by the irregular chain.

This encoding method, similarly to the baseheader_extension_headers encoding method above, is necessary to keep the definition of the header formats syntactically correct. It describes tunneling IP headers and their respective extension headers (i.e., all headers located before the innermost IP header) for CO headers (see Section 6.8.2).

6.6.3. inferred_udp_length

The decompressor infers the value of the UDP length field as being the sum of the UDP header length and the UDP payload length. The compressor must therefore ensure that the UDP length field is consistent with the length field(s) of preceding subheaders, i.e., there must not be any padding after the UDP payload that is covered by the IP Length.

This encoding method is also used for the UDP-Lite Checksum Coverage field when it behaves in the same manner as the UDP length field (i.e., when the checksum always covers the entire UDP-Lite payload).

6.6.4. inferred_ip_v4_header_checksum

This encoding method compresses the header checksum field of the IPv4 header. This checksum is defined in RFC 791 [RFC0791] as follows:

Header Checksum: 16 bits

A checksum on the header only. Since some header fields change (e.g., time to live), this is recomputed and verified at each point that the internet header is processed.

The checksum algorithm is:

The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero.

As described above, the header checksum protects individual hops from processing a corrupted header. As the data that this checksum protects is mostly compressed away and is instead taken from state stored in the context, this checksum becomes cumulative to the ROHC CRC. When using this encoding method, the checksum is recomputed by the decompressor.

The inferred_ip_v4_header_checksum encoding method thus compresses the header checksum field of the IPv4 header down to a size of zero bits, i.e., no bits are transmitted in compressed headers for this field. Using this encoding method, the decompressor infers the value of this field using the computation above.

The compressor MAY use the header checksum to validate the correctness of the header before compressing it, to avoid processing a corrupted header.

6.6.5. inferred_mine_header_checksum

This encoding method compresses the minimal encapsulation header checksum. This checksum is defined in RFC 2004 [RFC2004] as follows:

Header Checksum

The 16-bit one's complement of the one's complement sum of all 16-bit words in the minimal forwarding header. For purposes of computing the checksum, the value of the checksum field is 0. The IP header and IP payload (after the minimal forwarding header) are not included in this checksum computation.

The inferred_mine_header_checksum encoding method compresses the minimal encapsulation header checksum down to a size of zero bits, i.e., no bits are transmitted in compressed headers for this field. Using this encoding method, the decompressor infers the value of this field using the above computation.

The motivations for inferring this checksum are similar to the ones explained above in Section 6.6.4.

The compressor MAY use the minimal encapsulation header checksum to validate the correctness of the header before compressing it, to avoid processing a corrupted header.

6.6.6. inferred_ip_v4_length

This encoding method compresses the total length field of the IPv4 header. The total length field of the IPv4 header is defined in RFC 791 [RFC0791] as follows:

> Total Length: 16 bits
>
> > Total Length is the length of the datagram, measured in octets, including internet header and data. This field allows the length of a datagram to be up to 65,535 octets.

The inferred_ip_v4_length encoding method compresses the IPv4 header checksum down to a size of zero bits, i.e., no bits are transmitted in compressed headers for this field. Using this encoding method, the decompressor infers the value of this field by counting in octets the length of the entire packet after decompression.

6.6.7. inferred_ip_v6_length

This encoding method compresses the payload length field in the IPv6 header. This length field is defined in RFC 2460 [RFC2460] as follows:

> Payload Length: 16-bit unsigned integer
>
> > Length of the IPv6 payload, i.e., the rest of the packet following this IPv6 header, in octets. (Note that any extension headers present are considered part of the payload, i.e., included in the length count.)

The "inferred_ip_v6_length" encoding method compresses the payload length field of the IPv6 header down to a size of zero bits, i.e., no bits are transmitted in compressed headers for this field. Using this encoding method, the decompressor infers the value of this field by counting in octets the length of the entire packet after decompression.

IPv6 headers using the jumbo payload option of RFC 2675 [RFC2675] will not be compressible with this encoding method since the value of the payload length field does not match the length of the packet.

6.6.8. Scaled RTP Timestamp Compression

This section provides additional details on encodings used to scale the RTP timestamp, as defined in the formal notation in Section 6.8.2.4.

The RTP timestamp (TS) usually increases by a multiple of the RTP Sequence Number's (SN's) increase and is therefore a suitable candidate for scaled encoding. This scaling factor is labeled ts_stride in the definition of the profile in the formal notation. The compressor sets the scaling factor based on the change in TS with respect to the change in the RTP SN.

The default value of the scaling factor ts_stride is 160, as defined in Section 6.8.2.4. To use a different value for ts_stride, the compressor explicitly updates the value of ts_stride to the decompressor using one of the header formats that can carry this information.

When the compressor uses a scaling factor that is different than the default value of ts_stride, it can only use the new scaling factor once it has enough confidence that the decompressor has successfully calculated the residue (ts_offset) of the scaling function for the timestamp. The compressor achieves this by sending unscaled timestamp values, to allow the decompressor to establish the residue based on the current ts_stride. The compressor MAY send the unscaled timestamp in the same compressed header(s) used to establish the value of ts_stride.

Once the compressor has gained enough confidence that both the value of the scaling factor and the value of the residue have been established in the decompressor, the compressor can start compressing packets using the new scaling factor.

When the compressor detects that the residue (ts_offset) value has changed, it MUST NOT select a compressed header format that uses the scaled timestamp encoding before it has re-established the residue as described above.

When the value of the timestamp field wraps around, the value of the residue of the scaling function is likely to change. When this occurs, the compressor re-establishes the new residue value as described above.

If the decompressor receives a compressed header containing scaled timestamp bits while the ts_stride equals zero, it MUST NOT deliver the packet to upper layers and it SHOULD treat this as a CRC verification failure.

Whether or not the scaling is applied to the RTP TS field is up to the compressor implementation (i.e., the use of scaling is OPTIONAL), and is indicated by the tsc_indicator control field. In case scaling is applied to the RTP TS field, the value of ts_stride used by the compressor is up to the implementation. A value of ts_stride that is set to the expected increase in the RTP timestamp between consecutive unit increases of the RTP SN will provide the most gain for the scaled encoding. Other values may provide the same gain in some situations, but may reduce the gain in others.

When scaled timestamp encoding is used for header formats that do not transmit any lsb-encoded timestamp bits at all, the inferred_scaled_field encoding of Section 6.6.10 is used for encoding the timestamp.

6.6.9. timer_based_lsb

The timer-based compression encoding method, timer_based_lsb, compresses a field whose change pattern approximates a linear function of the time of day.

This encoding uses the local clock to obtain an approximation of the value that it encodes. The approximated value is then used as a reference value together with the num_lsbs_param least-significant bits received as the encoded value, where num_lsbs_param represents a number of bits that is sufficient to uniquely represent the encoded value in the presence of jitter between compression endpoints.

```
ts_scaled =:= timer_based_lsb(<time_stride_param>,
                              <num_lsbs_param>, <offset_param>)
```

The parameters "num_lsbs_param" and "offset_param" are the parameters to use for the lsb encoding, i.e., the number of least significant bits and the interpretation interval offset, respectively. The parameter "time_stride_param" represents the context value of the control field time_stride.

This encoding method always uses a scaled version of the field it compresses.

The value of the field is decoded by calculating an approximation of the scaled value, using:

```
tsc_ref_advanced = tsc_ref + (a_n - a_ref) / time_stride.
```

where:

- tsc_ref is a reference value of the scaled representation of the field.
- a_n is the arrival time associated with the value to decode.
- a_ref is the arrival time associated with the reference header.
- tsc_ref_advanced is an approximation of the scaled value of the field.

The lsb encoding is then applied using the num_lsbs_param bits received in the compressed header and the tsc_ref_advanced as "ref_value" (as per Section 4.11.5 of [RFC4997]).

Appendix B.3 provides an example of how the compressor can calculate jitter.

The control field time_stride controls whether or not the timer_based_lsb method is used in the CO header. The decompressor SHOULD send the CLOCK_RESOLUTION option with a zero value, if:

o it receives a non-zero time_stride value, and

o it has not previously sent a CLOCK_RESOLUTION feedback with a non-zero value.

This is to allow compression to recover from the case where a compressor erroneously activates timer-based compression.

The support and usage of timer-based compression is OPTIONAL for both the compressor and the decompressor; the compressor is not required to set the time_stride control field to a non-zero value when it has received a non-zero value for the CLOCK_RESOLUTION option.

6.6.10. inferred_scaled_field

The inferred_scaled_field encoding method encodes a field that is defined as changing in relation to the MSN, and for which the increase with respect to the MSN can be scaled by some scaling factor. This encoding method is used in compressed header formats that do not contain any bits for the scaled field. In this case, the decompressor infers the unscaled value of the scaled field from the MSN field. The unscaled value is calculated according to the following formula:

unscaled_value = delta_msn * stride + reference_unscaled_value

where "delta_msn" is the difference in MSN between the reference value of the MSN in the context and the value of the MSN decompressed

from this packet, "reference_unscaled_value" is the value of the field being scaled in the context, and "stride" is the scaling value for this field.

For example, when this encoding method is applied to the RTP timestamp in the RTP profile, the calculation above becomes:

```
timestamp = delta_msn * ts_stride + reference_timestamp
```

6.6.11. control_crc3_encoding

The control_crc3_encoding method provides a CRC calculated over a number of control fields. The definition of this encoding method is the same as for the "crc" encoding method specified in Section 4.11.6 of [RFC4997], with the difference being that the data covered by the CRC is given by a concatenated list of control fields.

In other words, the definition of the control_crc3_encoding method is equivalent to the following definition:

```
control_crc_encoding(ctrl_data_value, ctrl_data_length)
{
  UNCOMPRESSED {
  }

  COMPRESSED {
    control_crc3 =:=
      crc(3, 0x06, 0x07, ctrl_data_value, ctrl_data_length) [ 3 ];
  }
}
```

where the parameter "ctrl_data_value" binds to the concatenated values of the following control fields, in the order listed below:

- reorder_ratio, 2 bits padded with 6 MSB of zeroes
- ts_stride, 32 bits (only for profiles 0x0101 and 0x0107)
- time_stride, 32 bits (only for profiles 0x0101 and 0x0107)
- msn, 16 bits (not applicable for profiles 0x0101, 0x0103, and 0x0107)
- coverage_behavior, 2 bits padded with 6 MSB of zeroes (only for profiles 0x0107 and 0x0108)

- ip_id_behavior, one octet for each IP header in the compressible header chain starting from the outermost header. Each octet consists of 2 bits padded with 6 MSBs of zeroes.

The "ctrl_data_length" binds to the sum of the length of the control field(s) that are applicable to the specific profile.

The decompressor uses the resulting 3-bit CRC to validate the control fields that are updated by the co_common and co_repair header formats; this CRC cannot be used to verify the outcome of a decompression attempt.

This CRC protects the update of control fields, as the updated values are not always used to decompress the header that carries them and thus are not protected by the CRC-7 verification. This prevents impairments that could occur if the decompression of a co_common or of a co_repair succeeds and the decompressor sends positive feedback, while for some reason the control fields are incorrectly updated.

6.6.12. inferred_sequential_ip_id

This encoding method is used with a sequential IP-ID behavior (sequential or sequential byte-swapped) and when there are no coded IP-ID bits in the compressed header. In this case, the IP-ID offset from the MSN is constant, and the IP-ID increases by the same amount as the MSN (similar to the inferred_scaled_field encoding method).

The decompressor calculates the value for the IP-ID according to the following formula:

```
IP-ID = delta_msn + reference_IP_ID_value
```

where "delta_msn" is the difference between the reference value of the MSN in the context and the uncompressed value of the MSN associated to the compressed header, and where "reference_IP_ID_value" is the value of the IP-ID in the context. For swapped IP-ID behavior (i.e., when ip_id_behavior_innermost is set to IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED), "reference_IP_ID_value" and "IP-ID" are byte-swapped with regard to the corresponding fields in the context.

If the IP-ID behavior is random or zero, this encoding method does not update any fields.

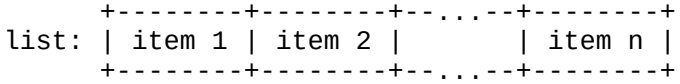
6.6.13. list_csrc(cc_value)

This encoding method compresses the list of RTP CSRC identifiers using list compression. This encoding establishes a content for the different CSRC identifiers (items) and a list describing the order in which they appear.

The compressor passes an argument (cc_value) to this encoding method: this is the value of the CC field taken from the RTP header. The decompressor is required to bind the value of this argument to the number of items in the list, which will allow the decompressor to correctly reconstruct the CC field.

6.6.13.1. List Compression

The CSRC identifiers in the uncompressed packet can be represented as an ordered list, whose order and presence are usually constant between packets. The generic structure of such a list is as follows:



When performing list compression on a CSRC list, each item is the uncompressed value of one CSRC identifier.

The basic principles of list-based compression are the following:

When initializing the context:

1) The complete representation of the list of CSRC identifiers is transmitted.

Then, once the context has been initialized:

2) When the list is unchanged, a compressed header that does not contain information about the list can be used.

3) When the list changes, a compressed list is sent in the compressed header, including a representation of its structure and order. Previously unknown items are sent uncompressed in the list, while previously known items are only represented by an index pointing to the item stored in the context.

6.6.13.2. Table-based Item Compression

The table-based item compression compresses individual items sent in compressed lists. The compressor assigns a unique identifier, "Index", to each item "Item" of a list.

Compressor Logic

The compressor conceptually maintains an item table containing all items, indexed using "Index". The (Index, Item) pair is sent together in compressed lists until the compressor gains enough confidence that the decompressor has observed the mapping between items and their respective index. Confidence is obtained from the reception of an acknowledgment from the decompressor, or by sending (Index, Item) pairs using the optimistic approach. Once confidence is obtained, the index alone is sent in compressed lists to indicate the presence of the item corresponding to this index.

The compressor MAY reset its item table upon receiving a negative acknowledgement.

The compressor MAY reassign an existing index to a new item by re-establishing the mapping using the procedure described above.

Decompressor Logic

The decompressor conceptually maintains an item table that contains all (Index, Item) pairs received. The item table is updated whenever an (Index, Item) pair is received and decompression is successful (CRC verification, or CRC-8 validation). The decompressor retrieves the item from the table whenever an Index is received without an accompanying Item.

If an index is received without an accompanying Item and the decompressor does not have any context for this index, the decompressor MUST NOT deliver the packet to upper layers.

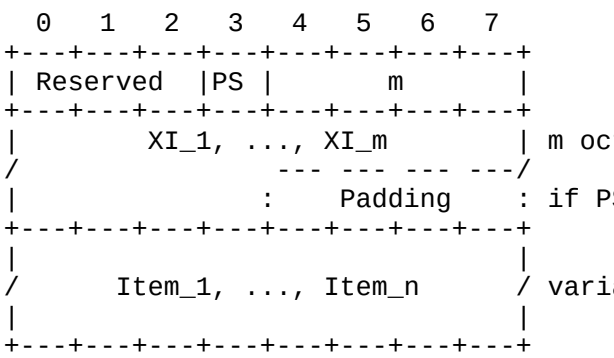
6.6.13.3. Encoding of Compressed Lists

Each item present in a compressed list is represented by:

- an Index into the table of items, and a presence bit indicating if a compressed representation of the item is present in the list.
- an item (if the presence bit is set).

If the presence bit is not set, the item must already be known by the decompressor.

A compressed list of items uses the following encoding:



Reserved: MUST be set to zero; otherwise, the decompressor MUST discard the packet.

PS: Indicates size of XI fields:

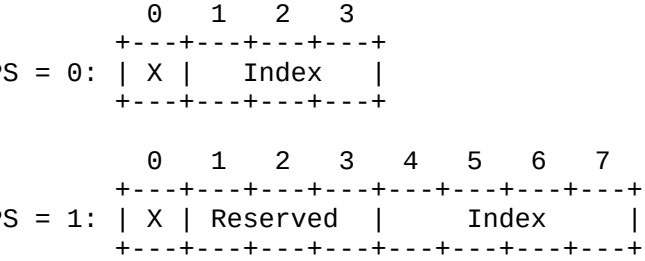
PS = 0 indicates 4-bit XI fields;

PS = 1 indicates 8-bit XI fields.

m: Number of XI item(s) in the compressed list. Also, the value of the cc_value argument of the list_csrc encoding (see Section 6.6.13).

XI_1, ..., XI_m: m XI items. Each XI represents one item in the list of items of the uncompressed header, in the same order as they appear in the uncompressed header.

The format of an XI item is as follows:



X: Indicates whether the item is present in the list:

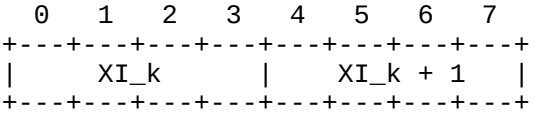
X = 1 indicates that the item corresponding to the Index is sent in the Item_1, ..., Item_n list;

X = 0 indicates that the item corresponding to the Index is not sent.

Reserved: MUST be set to zero; otherwise, the decompressor MUST discard the packet.

Index: An index into the item table. See Section 6.6.13.4

When 4-bit XI items are used, the XI items are placed in octets in the following manner:



Padding: A 4-bit Padding field is present when PS = 0 and the number of XIs is odd. The Padding field MUST be set to zero; otherwise, the decompressor MUST discard the packet.

Item 1, ..., item n: Each item corresponds to an XI with X = 1 in XI 1, ..., XI m. Each entry in the Item list is the uncompressed representation of one CSRC identifier.

6.6.13.4. Item Table Mappings

The item table for list compression is limited to 16 different items, since the RTP header can only carry at most 15 simultaneous CSRC identifiers. The effect of having more than 16 items in the item table will only cause a slight overhead to the compressor when items are swapped in/out of the item table.

6.6.13.5. Compressed Lists in Dynamic Chain

A compressed list that is part of the dynamic chain must have all of its list items present, i.e., all X-bits in the XI list MUST be set. All items previously established in the item table that are not present in the list decompressed from this packet MUST also be retained in the decompressor context.

6.7. Encoding Methods with External Parameters as Arguments

A number of encoding methods in Section 6.8.2.4 have one or more arguments for which the derivation of the parameter's value is outside the scope of the ROHC-FN [RFC4997] specification of the header formats.

The following is a list of encoding methods with external parameters as arguments, from Section 6.8.2.4:

- udp(profile_value, reorder_ratio_value)
- udp_lite(profile_value, reorder_ratio_value, coverage_behavior_value)
- esp(profile_value, reorder_ratio_value)
- rtp(profile_value, ts_stride_value, time_stride_value, reorder_ratio_value)
- ipv4(profile_value, is_innermost, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value))
- ipv6(profile_value, is_innermost, outer_ip_flag, reorder_ratio_value))
- iponly_baseheader(profile_value, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value)
- udp_baseheader(profile_value, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value)
- udplite_baseheader(profile_value, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value)
- esp_baseheader(profile_value, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value)
- rtp_baseheader(profile_value, ts_stride_value, time_stride_value, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value)
- udplite_rtp_baseheader(profile_value, ts_stride_value, time_stride_value, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value, coverage_behavior_value)

The following applies for all parameters listed below: At the compressor, the value of the parameter is set according to the recommendations for each parameter. At the decompressor, the value

of the parameter is set to undefined and will get bound by encoding methods, except where otherwise noted.

The following is a list of external arguments with their respective definition:

- profile_value:

  Set to the 16-bit number that identifies the profile used to compress this packet. When processing the static chain at the decompressor, this parameter is set to the value of the profile field in the IR header (see Section 6.8.1).

- reorder_ratio_value:

  Set to a 2-bit integer value, using one of the constants whose name begins with the prefix REORDERING_ and as defined in Section 6.8.2.4.

- ip_id_behavior_value:

  Set to a 2-bit integer value, using one of the constants whose name begins with the prefix IP_ID_BEHAVIOR_ and as defined in Section 6.8.2.4.

- coverage_behavior_value:

  Set to a 2-bit integer value, using one of the constants whose name begins with the prefix UDP_LITE_COVERAGE_ and as defined in Section 6.8.2.4.

- outer_ip_flag:

  This parameter is set to 1 if at least one of the TOS/TC or TTL/Hop Limit fields in outer IP headers has changed compared to their reference values in the context; otherwise, it is set to 0. This flag may only be set to 1 for the "co_common" header format in the different profiles.

- is_innermost:

  This boolean flag is set to 1 when processing the innermost of the compressible IP headers; otherwise, it is set to 0.

- ts_stride_value

  The value of this parameter should be set to the expected increase in the RTP Timestamp between consecutive RTP sequence numbers. The value selected is implementation-specific. See also Section 6.6.8.

- time_stride_value

  The value of this parameter should be set to the expected inter-arrival time between consecutive packets for the flow. The value selected is implementation-specific. This parameter MUST be set to zero, unless the compressor has received a feedback message with the CLOCK_RESOLUTION option set to a non-zero value. See also Section 6.6.9.

6.8. Header Formats

ROHCv2 profiles use two different header types: the Initialization and Refresh (IR) header type, and the Compressed header type (CO).

The CO header type defines a number of header formats: there are two sets of base header formats, with a few additional formats that are common to both sets.

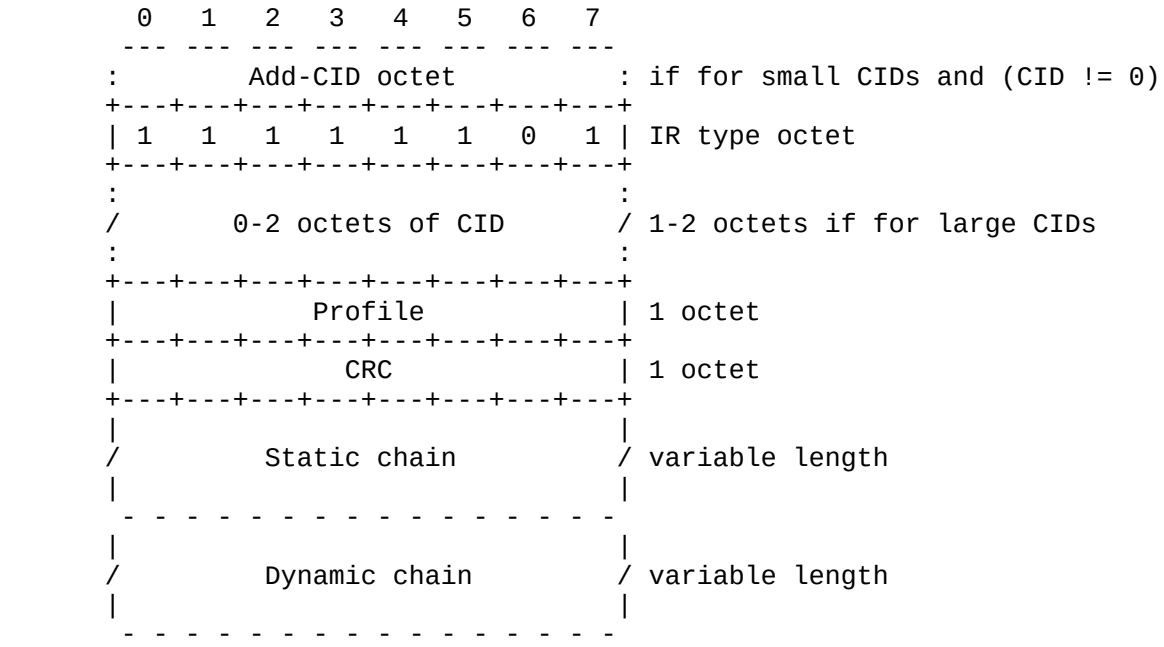
6.8.1. Initialization and Refresh Header Format (IR)

The IR header format uses the structure of the ROHC IR header as defined in Section 5.2.2.1 of [RFC4995].

Header type: IR

This header format communicates the static part and the dynamic part of the context.

The ROHCv2 IR header has the following format:



CRC: 8-bit CRC over the entire IR-header, including any CID fields and up until the end of the dynamic chain, using the polynomial defined in [RFC4995]. For purposes of computing the CRC, the CRC field is zero.

Static chain: See Section 6.5.

Dynamic chain: See Section 6.5.

6.8.2. Compressed Header Formats (CO)

6.8.2.1. Design Rationale for Compressed Base Headers

The compressed header formats are defined as two separate sets for each profile: one set for the headers where the innermost IP header contains a sequential IP-ID (either network byte order or byte-swapped), and one set for the headers without sequential IP-ID (either random, zero, or no IP-ID). There are also a number of common header formats shared between both sets. In the description below, the naming convention used for header formats that belong to the sequential set is to include "seq" in the name of the format, while similarly "rnd" is used for those that belong to the non-sequential set.

The design of the header formats is derived from the field behavior analysis found in Appendix A.

All of the compressed base headers transmit lsb-encoded MSN bits and a CRC.

The following header formats exist for all profiles defined in this document, and are common to both the sequential and the random header format sets:

- co_common: This format can be used to update the context when the established change pattern of a dynamic field changes, for any of the dynamic fields. However, not all dynamic fields are updated by conveying their uncompressed value; some fields can only be transmitted using a compressed representation. This format is especially useful when a rarely changing field needs to be updated. This format contains a set of flags to indicate what fields are present in the header, and its size can vary accordingly. This format is protected by a 7-bit CRC. It can update control fields, and it thus also carries a 3-bit CRC to protect those fields. This format is similar in purpose to the UOR-2-extension 3 format of [RFC3095].

- co_repair: This format can be used to update the context of all the dynamic fields by conveying their uncompressed value. This is especially useful when context damage is assumed (e.g., from the reception of a NACK) and a context repair is performed. This format is protected by a 7-bit CRC. It also carries a 3-bit CRC over the control fields that it can update. This format is similar in purpose to the IR-DYN format of [RFC3095] when performing context repairs.

- pt_0_crc3: This format conveys only the MSN; it can therefore only update the MSN and fields that are derived from the MSN, such as IP-ID and the RTP Timestamp (for applicable profiles). It is protected by a 3-bit CRC. This format is equivalent to the UO-0 header format in [RFC3095].

- pt_0_crc7: This format has the same properties as pt_0_crc3, but is instead protected by a 7-bit CRC and contains a larger amount of lsb-encoded MSN bits. This format is useful in environments where a high amount of reordering or a high-residual error rate can occur.

The following header format descriptions apply to profiles 0x0101 and 0x0107.

- pt_1_rnd: This format can convey changes to the MSN and to the RTP Marker bit, and it can update the RTP timestamp using scaled timestamp encoding. It is protected by a 3-bit CRC. It is similar in purpose to the UO-1 format in [RFC3095].

- pt_1_seq_id: This format can convey changes to the MSN and to the IP-ID. It is protected by a 3-bit CRC. It is similar in purpose to the UO-1-ID format in [RFC3095].

- pt_1_seq_ts: This format can convey changes to the MSN and to the RTP Marker bit, and it can update the RTP Timestamp using scaled timestamp encoding. It is protected by a 3-bit CRC. It is similar in purpose to the UO-1-TS format in [RFC3095].

- pt_2_rnd: This format can convey changes to the MSN, to the RTP Marker bit, and to the RTP Timestamp. It is protected by a 7-bit CRC. It is similar in purpose to the UOR-2 format in [RFC3095].

- pt_2_seq_id: This format can convey changes to the MSN and to the IP-ID. It is protected by a 7-bit CRC. It is similar in purpose to the UO-2-ID format in [RFC3095].

- pt_2_seq_ts: This format can convey changes to the MSN, to the RTP Marker bit and it can update the RTP Timestamp using scaled timestamp encoding. It is protected by a 7-bit CRC. It is similar in purpose to the UO-2-TS format in [RFC3095].

- pt_2_seq_both: This format can convey changes to both the RTP Timestamp and the IP-ID, in addition to the MSN and to the Marker bit. It is protected by a 7-bit CRC. It is similar in purpose to the UOR-2-ID extension 1 format in [RFC3095].

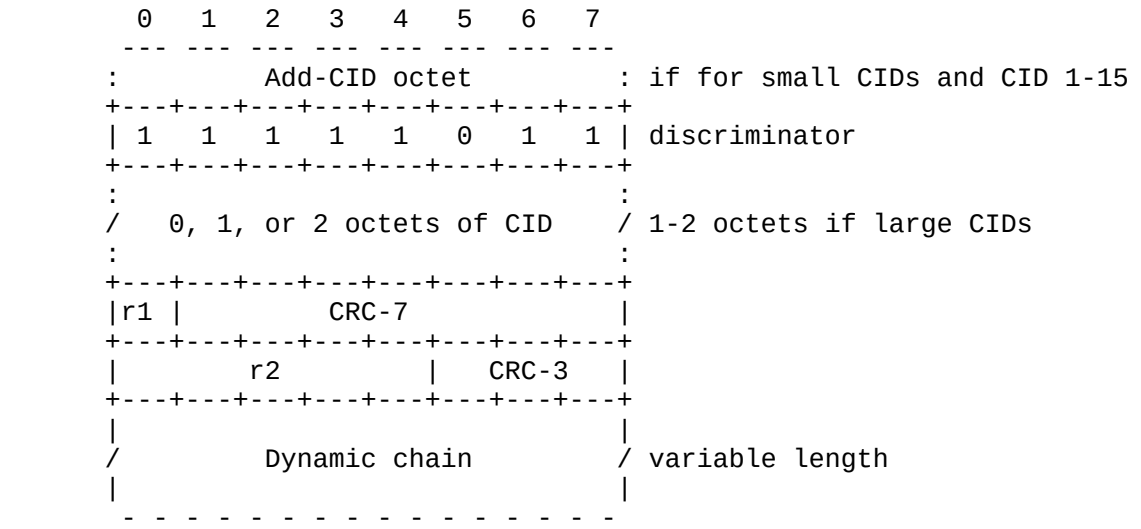
The following header format descriptions apply to profiles 0x0102, 0x0103, 0x0104, and 0x0108.

- pt_1_seq_id: This format can convey changes to the MSN and to the IP-ID. It is protected by a 7-bit CRC. It is similar in purpose to the UO-1-ID format in [RFC3095].

- pt_2_seq_id: This format can convey changes to the MSN and to the IP-ID. It is protected by a 7-bit CRC. It is similar in purpose to the UO-2-ID format in [RFC3095].

6.8.2.2. co_repair Header Format

The ROHCv2 co_repair header has the following format:



r1: MUST be set to zero; otherwise, the decompressor MUST discard the packet.

CRC-7: A 7-bit CRC over the entire uncompressed header, computed using the crc7 (data_value, data_length) encoding method defined in Section 6.8.2.4, where data_value corresponds to the entire uncompressed header chain and where data_length corresponds to the length of this header chain.

r2: MUST be set to zero; otherwise, the decompressor MUST discard the packet.

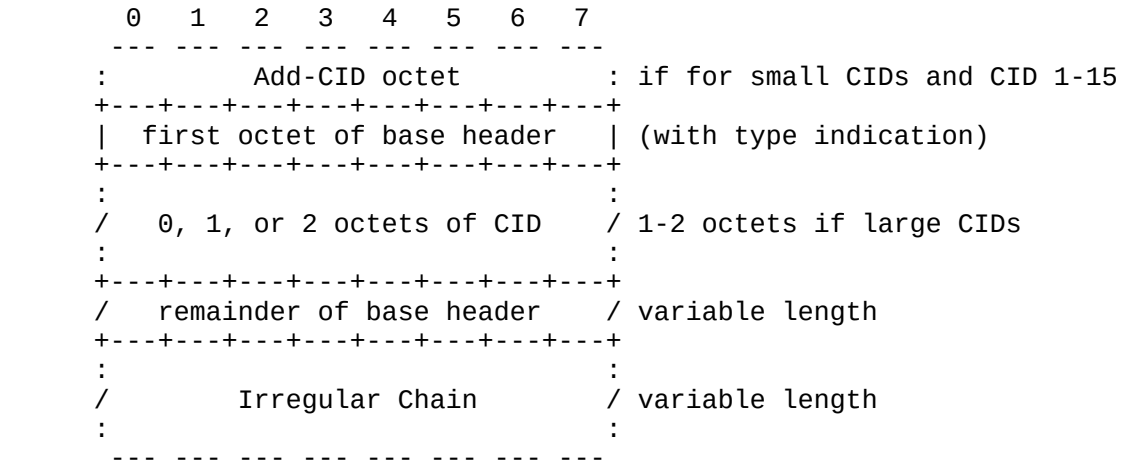
CRC-3: Encoded using the control_crc3_encoding method defined in Section 6.6.11.

Dynamic chain: See Section 6.5.

6.8.2.3. General CO Header Format

The CO header format communicates irregularities in the packet header. All CO formats carry a CRC and can update the context. All CO header formats use the general format defined in this section, with the exception of the co_repair format, which is defined in Section 6.8.2.2.

The general format for a compressed header is as follows:



The base header in the figure above is the compressed representation of the innermost IP header and other header(s), if any, in the uncompressed packet. The base header formats are defined in Section 6.8.2.4. In the formal description of the header formats, the base header for each profile is labeled <profile_name>_baseheader, where <profile_name> is defined in the following table:

Profile number	profile_name
0x0101	rtp
0x0102	udp
0x0103	esp
0x0104	ip
0x0107	udplite_rtp
0x0108	udplite

6.8.2.4. Header Formats in ROHC-FN

This section defines the complete set of base header formats for ROHCv2 profiles. The base header formats are defined using the ROHC Formal Notation [RFC4997].

```
// NOTE: The irregular, static, and dynamic chains (see Section 6.5)
// are defined across multiple encoding methods and are embodied
// in the correspondingly named formats within those encoding
// methods.  In particular, note that the static and dynamic
// chains ordinarily go together.  The uncompressed fields are
// defined across these two formats combined, rather than in one
// or the other of them.  The irregular chain items are likewise
// combined with a baseheader format.

////////////////////////////////////////
// Constants
////////////////////////////////////////

// IP-ID behavior constants
IP_ID_BEHAVIOR_SEQUENTIAL         = 0;
IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED = 1;
IP_ID_BEHAVIOR_RANDOM             = 2;
IP_ID_BEHAVIOR_ZERO               = 3;

// UDP-lite checksum coverage behavior constants
UDP_LITE_COVERAGE_INFERRED  = 0;
UDP_LITE_COVERAGE_STATIC    = 1;
UDP_LITE_COVERAGE_IRREGULAR = 2;
// The value 3 is reserved and cannot be used for coverage behavior

// Variable reordering offset
REORDERING_NONE          = 0;
REORDERING_QUARTER       = 1;
REORDERING_HALF          = 2;
REORDERING_THREEQUARTERS = 3;

// Profile names and versions
PROFILE_RTP_0101     = 0x0101;
PROFILE_UDP_0102     = 0x0102;
PROFILE_ESP_0103     = 0x0103;
PROFILE_IP_0104      = 0x0104;
PROFILE_RTP_0107     = 0x0107; // With UDP-LITE
PROFILE_UDPLITE_0108 = 0x0108; // Without RTP

// Default values for RTP timestamp encoding
TS_STRIDE_DEFAULT    = 160;
TIME_STRIDE_DEFAULT  = 0;

////////////////////////////////////////
// Global control fields
////////////////////////////////////////

CONTROL {
```

```
  profile                                    [ 16 ];
  msn                                        [ 16 ];
  reorder_ratio                              [  2 ];
  // ip_id fields are for innermost IP header only
  ip_id_offset                               [ 16 ];
  ip_id_behavior_innermost                   [  2 ];
  // The following are only used in RTP-based profiles
  ts_stride                                  [ 32 ];
  time_stride                                [ 32 ];
  ts_scaled                                  [ 32 ];
  ts_offset                                  [ 32 ];
  // UDP-lite-based profiles only
  coverage_behavior                          [  2 ];
}

////////////////////////////////////////////
// Encoding methods not specified in FN syntax:
////////////////////////////////////////////

baseheader_extension_headers       "defined in Section 6.6.1";
baseheader_outer_headers           "defined in Section 6.6.2";
control_crc3_encoding              "defined in Section 6.6.11";
inferred_ip_v4_header_checksum     "defined in Section 6.6.4";
inferred_ip_v4_length              "defined in Section 6.6.6";
inferred_ip_v6_length              "defined in Section 6.6.7";
inferred_mine_header_checksum      "defined in Section 6.6.5";
inferred_scaled_field              "defined in Section 6.6.10";
inferred_sequential_ip_id          "defined in Section 6.6.12";
inferred_udp_length                "defined in Section 6.6.3";
list_csrc(cc_value)                "defined in Section 6.6.13";
timer_based_lsb(time_stride, k, p) "defined in Section 6.6.9";

//////////////////////////////////////////
// General encoding methods
//////////////////////////////////////////

static_or_irreg(flag, width)
{
  UNCOMPRESSED {
    field [ width ];
  }

  COMPRESSED irreg_enc {
    ENFORCE(flag == 1);
    field =:= irregular(width) [ width ];
  }

  COMPRESSED static_enc {
```

```
    ENFORCE(flag == 0);
    field =:= static [ 0 ];
  }
}

optional_32(flag)
{
  UNCOMPRESSED {
    item [ 0, 32 ];
  }

  COMPRESSED present {
    ENFORCE(flag == 1);
    item =:= irregular(32) [ 32 ];
  }

  COMPRESSED not_present {
    ENFORCE(flag == 0);
    item =:= compressed_value(0, 0) [ 0 ];
  }
}

// Send the entire value, or keep previous value
sdvl_or_static(flag)
{
  UNCOMPRESSED {
    field [ 32 ];
  }

  COMPRESSED present_7bit {
    ENFORCE(flag == 1);
    ENFORCE(field.UVALUE < 2^7);
    ENFORCE(field.CVALUE == field.UVALUE);
    discriminator =:= '0' [ 1 ];
    field                 [ 7 ];
  }

  COMPRESSED present_14bit {
    ENFORCE(flag == 1);
    ENFORCE(field.UVALUE < 2^14);
    ENFORCE(field.CVALUE == field.UVALUE);
    discriminator =:= '10'   [  2 ];
    field                    [ 14 ];
  }

  COMPRESSED present_21bit {
    ENFORCE(flag == 1);
    ENFORCE(field.UVALUE < 2^21);
```

```
    ENFORCE(field.CVALUE == field.UVALUE);
    discriminator =:= '110'  [  3 ];
    field                    [ 21 ];
  }

  COMPRESSED present_28bit {
    ENFORCE(flag == 1);
    ENFORCE(field.UVALUE < 2^28);
    ENFORCE(field.CVALUE == field.UVALUE);
    discriminator =:= '1110'  [  4 ];
    field                     [ 28 ];
  }

  COMPRESSED present_32bit {
    ENFORCE(flag == 1);
    ENFORCE(field.CVALUE == field.UVALUE);
    discriminator =:= '11111111'  [  8 ];
    field                         [ 32 ];
  }

  COMPRESSED not_present {
    ENFORCE(flag == 0);
    field =:= static;
  }
}

// Send the entire value, or revert to default value
sdvl_or_default(flag, default_value)
{
  UNCOMPRESSED {
    field [ 32 ];
  }

  COMPRESSED present_7bit {
    ENFORCE(flag == 1);
    ENFORCE(field.UVALUE < 2^7);
    ENFORCE(field.CVALUE == field.UVALUE);
    discriminator =:= '0' [ 1 ];
    field                 [ 7 ];
  }

  COMPRESSED present_14bit {
    ENFORCE(flag == 1);
    ENFORCE(field.UVALUE < 2^14);
    ENFORCE(field.CVALUE == field.UVALUE);
    discriminator =:= '10'   [  2 ];
    field                    [ 14 ];
  }
```

```
  COMPRESSED present_21bit {
    ENFORCE(flag == 1);
    ENFORCE(field.UVALUE < 2^21);
    ENFORCE(field.CVALUE == field.UVALUE);
    discriminator =:= '110'  [  3 ];
    field                    [ 21 ];
  }

  COMPRESSED present_28bit {
    ENFORCE(flag == 1);
    ENFORCE(field.UVALUE < 2^28);
    ENFORCE(field.CVALUE == field.UVALUE);
    discriminator =:= '1110'  [  4 ];
    field                     [ 28 ];
  }

  COMPRESSED present_32bit {
    ENFORCE(flag == 1);
    ENFORCE(field.CVALUE == field.UVALUE);
    discriminator =:= '11111111'  [  8 ];
    field                         [ 32 ];
  }

  COMPRESSED not_present {
    ENFORCE(flag == 0);
    field =:= uncompressed_value(32, default_value);
  }
}

lsb_7_or_31
{
  UNCOMPRESSED {
    item [ 32 ];
  }

  COMPRESSED lsb_7 {
    discriminator =:= '0'                       [  1 ];
    item          =:= lsb(7, ((2^7) / 4) - 1)   [  7 ];
  }

  COMPRESSED lsb_31 {
    discriminator =:= '1'                       [  1 ];
    item          =:= lsb(31, ((2^31) / 4) - 1) [ 31 ];
  }
}

crc3(data_value, data_length)
{
```

```
  UNCOMPRESSED {
  }
  COMPRESSED {
    crc_value =:= crc(3, 0x06, 0x07, data_value, data_length) [ 3 ];
  }
}

crc7(data_value, data_length)
{
  UNCOMPRESSED {
  }

  COMPRESSED {
    crc_value =:= crc(7, 0x79, 0x7f, data_value, data_length) [ 7 ];
  }
}

// Encoding method for updating a scaled field and its associated
// control fields.  Should be used both when the value is scaled
// or unscaled in a compressed format.
// Does not have an uncompressed side.
field_scaling(stride_value, scaled_value, unscaled_value, residue_value)
{
  UNCOMPRESSED {
    // Nothing
  }

  COMPRESSED no_scaling {
    ENFORCE(stride_value == 0);
    ENFORCE(residue_value == unscaled_value);
    ENFORCE(scaled_value == 0);
  }

  COMPRESSED scaling_used {
    ENFORCE(stride_value != 0);
    ENFORCE(residue_value == (unscaled_value % stride_value));
    ENFORCE(unscaled_value ==
            scaled_value * stride_value + residue_value);
  }
}

//////////////////////////////////////////
// IPv6 Destination options header
//////////////////////////////////////////

ip_dest_opt
{
  UNCOMPRESSED {
```

```
    next_header [ 8 ];
    length      [ 8 ];
    value       [ length.UVALUE * 64 + 48 ];
  }

  DEFAULT {
    length      =:= static;
    next_header =:= static;
    value       =:= static;
  }

  COMPRESSED dest_opt_static {
    next_header =:= irregular(8) [ 8 ];
    length      =:= irregular(8) [ 8 ];
  }

  COMPRESSED dest_opt_dynamic {
    value =:=
      irregular(length.UVALUE * 64 + 48) [ length.UVALUE * 64 + 48 ];
  }

  COMPRESSED dest_opt_irregular {
  }

}

//////////////////////////////////////////
// IPv6 Hop-by-Hop options header
//////////////////////////////////////////

ip_hop_opt
{
  UNCOMPRESSED {
    next_header [ 8 ];
    length      [ 8 ];
    value       [ length.UVALUE * 64 + 48 ];
  }

  DEFAULT {
    length      =:= static;
    next_header =:= static;
    value       =:= static;
  }

  COMPRESSED hop_opt_static {
    next_header =:= irregular(8) [ 8 ];
    length      =:= irregular(8) [ 8 ];
  }
```

```
  COMPRESSED hop_opt_dynamic {
    value =:=
      irregular(length.UVALUE*64+48) [ length.UVALUE * 64 + 48 ];
  }

  COMPRESSED hop_opt_irregular {
  }

}

//////////////////////////////////////////
// IPv6 Routing header
//////////////////////////////////////////

ip_rout_opt
{
  UNCOMPRESSED {
    next_header [ 8 ];
    length      [ 8 ];
    value       [ length.UVALUE * 64 + 48 ];
  }

  DEFAULT {
    length      =:= static;
    next_header =:= static;
    value       =:= static;
  }

  COMPRESSED rout_opt_static {
    next_header =:= irregular(8)                  [ 8 ];
    length      =:= irregular(8)                  [ 8 ];
    value       =:=
      irregular(length.UVALUE*64+48) [ length.UVALUE * 64 + 48 ];
  }

  COMPRESSED rout_opt_dynamic {
  }

  COMPRESSED rout_opt_irregular {
  }
}

//////////////////////////////////////////
// GRE Header
//////////////////////////////////////////

optional_lsb_7_or_31(flag)
{
```

```
  UNCOMPRESSED {
    item [ 0, 32 ];
  }

  COMPRESSED present {
    ENFORCE(flag == 1);
    item =:= lsb_7_or_31 [ 8, 32 ];
  }

  COMPRESSED not_present {
    ENFORCE(flag == 0);
    item =:= compressed_value(0, 0) [ 0 ];
  }
}

optional_checksum(flag_value)
{
  UNCOMPRESSED {
    value     [ 0, 16 ];
    reserved1 [ 0, 16 ];
  }

  COMPRESSED cs_present {
    ENFORCE(flag_value == 1);
    value     =:= irregular(16)              [ 16 ];
    reserved1 =:= uncompressed_value(16, 0) [  0 ];
  }

  COMPRESSED not_present {
    ENFORCE(flag_value == 0);
    value     =:= compressed_value(0, 0) [ 0 ];
    reserved1 =:= compressed_value(0, 0) [ 0 ];
  }
}

gre_proto
{
  UNCOMPRESSED {
    protocol [ 16 ];
  }

  COMPRESSED ether_v4 {
    discriminator =:= '0'                              [ 1 ];
    protocol      =:= uncompressed_value(16, 0x0800) [ 0 ];
  }

  COMPRESSED ether_v6 {
    discriminator =:= '1'                              [ 1 ];
```

```
    protocol      =:= uncompressed_value(16, 0x86DD) [ 0 ];
  }
}

gre
{
  UNCOMPRESSED {
    c_flag                                  [  1 ];
    r_flag    =:= uncompressed_value(1, 0) [  1 ];
    k_flag                                  [  1 ];
    s_flag                                  [  1 ];
    reserved0 =:= uncompressed_value(9, 0) [  9 ];
    version   =:= uncompressed_value(3, 0) [  3 ];
    protocol                                [ 16 ];
    checksum_and_res                        [ 0, 32 ];
    key                                     [ 0, 32 ];
    sequence_number                         [ 0, 32 ];
  }

  DEFAULT {
    c_flag           =:= static;
    k_flag           =:= static;
    s_flag           =:= static;
    protocol         =:= static;
    key              =:= static;
    sequence_number  =:= static;
  }

  COMPRESSED gre_static {
    ENFORCE((c_flag.UVALUE == 1 && checksum_and_res.ULENGTH == 32)
            || checksum_and_res.ULENGTH == 0);
    ENFORCE((s_flag.UVALUE == 1 && sequence_number.ULENGTH == 32)
            || sequence_number.ULENGTH == 0);
    protocol =:= gre_proto                  [ 1 ];
    c_flag   =:= irregular(1)               [ 1 ];
    k_flag   =:= irregular(1)               [ 1 ];
    s_flag   =:= irregular(1)               [ 1 ];
    padding  =:= compressed_value(4, 0)     [ 4 ];
    key      =:= optional_32(k_flag.UVALUE) [ 0, 32 ];
  }

  COMPRESSED gre_dynamic {
    checksum_and_res =:=
      optional_checksum(c_flag.UVALUE)              [ 0, 16 ];
    sequence_number  =:= optional_32(s_flag.UVALUE) [ 0, 32 ];
  }

  COMPRESSED gre_irregular {
```

```
    checksum_and_res =:= optional_checksum(c_flag.UVALUE) [ 0, 16 ];
    sequence_number  =:=
      optional_lsb_7_or_31(s_flag.UVALUE)          [ 0, 8, 32 ];
  }

}

//////////////////////////////////////////
// MINE header
//////////////////////////////////////////

mine
{
  UNCOMPRESSED {
    next_header [  8 ];
    s_bit       [  1 ];
    res_bits    [  7 ];
    checksum    [ 16 ];
    orig_dest   [ 32 ];
    orig_src    [ 0, 32 ];
  }

  DEFAULT {
    next_header =:= static;
    s_bit       =:= static;
    res_bits    =:= static;
    checksum    =:= inferred_mine_header_checksum;
    orig_dest   =:= static;
    orig_src    =:= static;
  }

  COMPRESSED mine_static {
    next_header =:= irregular(8)              [  8 ];
    s_bit       =:= irregular(1)              [  1 ];
    // Reserved bits are included to achieve byte-alignment
    res_bits    =:= irregular(7)              [  7 ];
    orig_dest   =:= irregular(32)             [ 32 ];
    orig_src    =:= optional_32(s_bit.UVALUE) [ 0, 32 ];
  }

  COMPRESSED mine_dynamic {
  }

  COMPRESSED mine_irregular {
  }
}

//////////////////////////////////////////
```

```
// Authentication Header (AH)
/////////////////////////////////////////

ah
{
  UNCOMPRESSED {
    next_header                              [  8 ];
    length                                   [  8 ];
    res_bits =:= uncompressed_value(16, 0) [ 16 ];
    spi                                      [ 32 ];
    sequence_number                          [ 32 ];
    icv                  [ length.UVALUE*32-32 ];
  }

  DEFAULT {
    next_header     =:= static;
    length          =:= static;
    spi             =:= static;
    sequence_number =:= static;
  }

  COMPRESSED ah_static {
    next_header =:= irregular(8)      [  8 ];
    length      =:= irregular(8)      [  8 ];
    spi         =:= irregular(32)     [ 32 ];
  }

  COMPRESSED ah_dynamic {
    sequence_number =:= irregular(32) [ 32 ];
    icv       =:=
      irregular(length.UVALUE*32-32)  [ length.UVALUE*32-32 ];
  }

  COMPRESSED ah_irregular {
    sequence_number =:= lsb_7_or_31   [ 8, 32 ];
    icv       =:=
      irregular(length.UVALUE*32-32)  [ length.UVALUE*32-32 ];
  }

}

/////////////////////////////////////////
// IPv6 Header
/////////////////////////////////////////

fl_enc
{
  UNCOMPRESSED {
```

```
    flow_label [ 20 ];
  }

  COMPRESSED fl_zero {
    discriminator =:= '0'                        [ 1 ];
    flow_label    =:= uncompressed_value(20, 0) [ 0 ];
    reserved      =:= '0000'                     [ 4 ];
  }

  COMPRESSED fl_non_zero {
    discriminator =:= '1'           [  1 ];
    flow_label    =:= irregular(20) [ 20 ];
  }
}

ipv6(profile_value, is_innermost, outer_ip_flag, reorder_ratio_value)
{
  UNCOMPRESSED {
    version         =:= uncompressed_value(4, 6) [   4 ];
    tos_tc                                       [   8 ];
    flow_label                                   [  20 ];
    payload_length                               [  16 ];
    next_header                                  [   8 ];
    ttl_hopl                                     [   8 ];
    src_addr                                     [ 128 ];
    dst_addr                                     [ 128 ];
  }

  CONTROL {
    ENFORCE(profile == profile_value);
    ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
    ENFORCE(innermost_ip.UVALUE == is_innermost);
    innermost_ip [ 1 ];
  }

  DEFAULT {
    tos_tc         =:= static;
    flow_label     =:= static;
    payload_length =:= inferred_ip_v6_length;
    next_header    =:= static;
    ttl_hopl       =:= static;
    src_addr       =:= static;
    dst_addr       =:= static;
  }

  COMPRESSED ipv6_static {
    version_flag        =:= '1'              [   1 ];
    innermost_ip        =:= irregular(1)     [   1 ];
```

```
    reserved            =:= '0'              [   1 ];
    flow_label          =:= fl_enc           [ 5, 21 ];
    next_header         =:= irregular(8)     [   8 ];
    src_addr            =:= irregular(128)   [ 128 ];
    dst_addr            =:= irregular(128)   [ 128 ];
  }

  COMPRESSED ipv6_endpoint_dynamic {
    ENFORCE((is_innermost == 1) &&
            (profile_value == PROFILE_IP_0104));
    tos_tc        =:= irregular(8)           [  8 ];
    ttl_hopl      =:= irregular(8)           [  8 ];
    reserved      =:= compressed_value(6, 0) [  6 ];
    reorder_ratio =:= irregular(2)           [  2 ];
    msn           =:= irregular(16)          [ 16 ];
  }

  COMPRESSED ipv6_regular_dynamic {
    ENFORCE((is_innermost == 0) ||
            (profile_value != PROFILE_IP_0104));
    tos_tc       =:= irregular(8) [ 8 ];
    ttl_hopl     =:= irregular(8) [ 8 ];
  }

  COMPRESSED ipv6_outer_irregular {
    ENFORCE(is_innermost == 0);
    tos_tc       =:=
        static_or_irreg(outer_ip_flag, 8) [ 0, 8 ];
    ttl_hopl     =:=
        static_or_irreg(outer_ip_flag, 8) [ 0, 8 ];
  }

  COMPRESSED ipv6_innermost_irregular {
    ENFORCE(is_innermost == 1);
  }

}

////////////////////////////////////////////
// IPv4 Header
////////////////////////////////////////////

ip_id_enc_dyn(behavior)
{
  UNCOMPRESSED {
    ip_id [ 16 ];
  }
```

```
  COMPRESSED ip_id_seq {
    ENFORCE((behavior == IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (behavior == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    ENFORCE(ip_id_offset.UVALUE == ip_id.UVALUE - msn.UVALUE);
    ip_id =:= irregular(16) [ 16 ];
  }

  COMPRESSED ip_id_random {
    ENFORCE(behavior == IP_ID_BEHAVIOR_RANDOM);
    ip_id =:= irregular(16) [ 16 ];
  }

  COMPRESSED ip_id_zero {
    ENFORCE(behavior == IP_ID_BEHAVIOR_ZERO);
    ip_id =:= uncompressed_value(16, 0) [ 0 ];
  }
}

ip_id_enc_irreg(behavior)
{
  UNCOMPRESSED {
    ip_id [ 16 ];
  }

  COMPRESSED ip_id_seq {
    ENFORCE(behavior == IP_ID_BEHAVIOR_SEQUENTIAL);
  }

  COMPRESSED ip_id_seq_swapped {
    ENFORCE(behavior == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED);
  }

  COMPRESSED ip_id_rand {
    ENFORCE(behavior == IP_ID_BEHAVIOR_RANDOM);
    ip_id =:= irregular(16) [ 16 ];
  }

  COMPRESSED ip_id_zero {
    ENFORCE(behavior == IP_ID_BEHAVIOR_ZERO);
    ip_id =:= uncompressed_value(16, 0) [ 0 ];
  }
}

ipv4(profile_value, is_innermost, outer_ip_flag, ip_id_behavior_value,
  reorder_ratio_value)
{
  UNCOMPRESSED {
    version     =:= uncompressed_value(4, 4)       [  4 ];
```

```
    hdr_length  =:= uncompressed_value(4, 5)       [  4 ];
    tos_tc                                         [  8 ];
    length      =:= inferred_ip_v4_length          [ 16 ];
    ip_id                                          [ 16 ];
    rf          =:= uncompressed_value(1, 0)       [  1 ];
    df                                             [  1 ];
    mf          =:= uncompressed_value(1, 0)       [  1 ];
    frag_offset =:= uncompressed_value(13, 0)      [ 13 ];
    ttl_hopl                                       [  8 ];
    protocol                                       [  8 ];
    checksum    =:= inferred_ip_v4_header_checksum [ 16 ];
    src_addr                                       [ 32 ];
    dst_addr                                       [ 32 ];
  }

  CONTROL {
    ENFORCE(profile == profile_value);
    ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
    ENFORCE(innermost_ip.UVALUE == is_innermost);
    ip_id_behavior_outer [ 2 ];
    innermost_ip [ 1 ];
  }

  DEFAULT {
    tos_tc               =:= static;
    df                   =:= static;
    ttl_hopl             =:= static;
    protocol             =:= static;
    src_addr             =:= static;
    dst_addr             =:= static;
    ip_id_behavior_outer =:= static;
  }

  COMPRESSED ipv4_static {
    version_flag        =:= '0'                    [  1 ];
    innermost_ip        =:= irregular(1)           [  1 ];
    reserved            =:= '000000'               [  6 ];
    protocol            =:= irregular(8)           [  8 ];
    src_addr            =:= irregular(32)          [ 32 ];
    dst_addr            =:= irregular(32)          [ 32 ];
  }

  COMPRESSED ipv4_endpoint_innermost_dynamic {
    ENFORCE((is_innermost == 1) && (profile_value == PROFILE_IP_0104));
    ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
    reserved       =:= '000'                                [  3 ];
    reorder_ratio  =:= irregular(2)                         [  2 ];
    df             =:= irregular(1)                         [  1 ];
```

```
    ip_id_behavior_innermost =:= irregular(2)               [  2 ];
    tos_tc         =:= irregular(8)                         [  8 ];
    ttl_hopl       =:= irregular(8)                         [  8 ];
    ip_id =:= ip_id_enc_dyn(ip_id_behavior_innermost.UVALUE) [ 0, 16 ];
    msn            =:= irregular(16)                        [ 16 ];
  }

  COMPRESSED ipv4_regular_innermost_dynamic {
    ENFORCE((is_innermost == 1) && (profile_value != PROFILE_IP_0104));
    ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
    reserved       =:= '00000'                              [ 5 ];
    df             =:= irregular(1)                         [ 1 ];
    ip_id_behavior_innermost =:= irregular(2)               [ 2 ];
    tos_tc         =:= irregular(8)                         [ 8 ];
    ttl_hopl       =:= irregular(8)                         [ 8 ];
    ip_id =:= ip_id_enc_dyn(ip_id_behavior_innermost.UVALUE) [ 0, 16 ];
  }

  COMPRESSED ipv4_outer_dynamic {
    ENFORCE(is_innermost == 0);
    ENFORCE(ip_id_behavior_outer.UVALUE == ip_id_behavior_value);
    reserved       =:= '00000'                            [ 5 ];
    df             =:= irregular(1)                       [ 1 ];
    ip_id_behavior_outer =:=     irregular(2)             [ 2 ];
    tos_tc         =:= irregular(8)                       [ 8 ];
    ttl_hopl       =:= irregular(8)                       [ 8 ];
    ip_id =:= ip_id_enc_dyn(ip_id_behavior_outer.UVALUE)  [ 0, 16 ];
  }

  COMPRESSED ipv4_outer_irregular {
    ENFORCE(is_innermost == 0);
    ip_id    =:=
      ip_id_enc_irreg(ip_id_behavior_outer.UVALUE)     [ 0, 16 ];
    tos_tc   =:= static_or_irreg(outer_ip_flag, 8)    [  0, 8 ];
    ttl_hopl =:= static_or_irreg(outer_ip_flag, 8)    [  0, 8 ];
  }

  COMPRESSED ipv4_innermost_irregular {
    ENFORCE(is_innermost == 1);
    ip_id =:=
      ip_id_enc_irreg(ip_id_behavior_innermost.UVALUE)  [ 0, 16 ];
  }

}

//////////////////////////////////////////
// UDP Header
//////////////////////////////////////////
```

```
udp(profile_value, reorder_ratio_value)
{
  UNCOMPRESSED {
    ENFORCE((profile_value == PROFILE_RTP_0101) ||
            (profile_value == PROFILE_UDP_0102));
    src_port                          [ 16 ];
    dst_port                          [ 16 ];
    udp_length =:= inferred_udp_length [ 16 ];
    checksum                          [ 16 ];
  }

  CONTROL {
    ENFORCE(profile == profile_value);
    ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
    checksum_used [ 1 ];
  }

  DEFAULT {
    src_port      =:= static;
    dst_port      =:= static;
    checksum_used =:= static;
  }

  COMPRESSED udp_static {
    src_port   =:= irregular(16) [ 16 ];
    dst_port   =:= irregular(16) [ 16 ];
  }

  COMPRESSED udp_endpoint_dynamic {
    ENFORCE(profile_value == PROFILE_UDP_0102);
    ENFORCE(profile == PROFILE_UDP_0102);
    ENFORCE(checksum_used.UVALUE == (checksum.UVALUE != 0));
    checksum      =:= irregular(16)          [ 16 ];
    msn           =:= irregular(16)          [ 16 ];
    reserved      =:= compressed_value(6, 0) [  6 ];
    reorder_ratio =:= irregular(2)           [  2 ];
  }

  COMPRESSED udp_regular_dynamic {
    ENFORCE(profile_value == PROFILE_RTP_0101);
    ENFORCE(checksum_used.UVALUE == (checksum.UVALUE != 0));
    checksum =:= irregular(16) [ 16 ];
  }

  COMPRESSED udp_zero_checksum_irregular {
    ENFORCE(checksum_used.UVALUE == 0);
    checksum =:= uncompressed_value(16, 0)   [ 0 ];
  }
```

```
  COMPRESSED udp_with_checksum_irregular {
    ENFORCE(checksum_used.UVALUE == 1);
    checksum =:= irregular(16) [ 16 ];
  }

}

//////////////////////////////////////////
// RTP Header
//////////////////////////////////////////

csrc_list_dynchain(presence, cc_value)
{
  UNCOMPRESSED {
    csrc_list;
  }

  COMPRESSED no_list {
    ENFORCE(cc_value == 0);
    ENFORCE(presence == 0);
    csrc_list =:= uncompressed_value(0, 0) [ 0 ];
  }

  COMPRESSED list_present {
    ENFORCE(presence == 1);
    csrc_list =:= list_csrc(cc_value) [ VARIABLE ];
  }
}

rtp(profile_value, ts_stride_value, time_stride_value,
    reorder_ratio_value)
{
  UNCOMPRESSED {
    ENFORCE((profile_value == PROFILE_RTP_0101) ||
            (profile_value == PROFILE_RTP_0107));
    rtp_version =:= uncompressed_value(2, 0) [  2 ];
    pad_bit                                  [  1 ];
    extension                                [  1 ];
    cc                                       [  4 ];
    marker                                   [  1 ];
    payload_type                             [  7 ];
    sequence_number                          [ 16 ];
    timestamp                                [ 32 ];
    ssrc                                     [ 32 ];
    csrc_list                                [ cc.UVALUE * 32 ];
  }

  CONTROL {
```

```
    ENFORCE(profile == profile_value);
    ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
    ENFORCE(time_stride_value == time_stride.UVALUE);
    ENFORCE(ts_stride_value == ts_stride.UVALUE);
    dummy_field =:= field_scaling(ts_stride.UVALUE,
      ts_scaled.UVALUE, timestamp.UVALUE, ts_offset.UVALUE) [ 0 ];
  }

  INITIAL {
    ts_stride     =:= uncompressed_value(32, TS_STRIDE_DEFAULT);
    time_stride   =:= uncompressed_value(32, TIME_STRIDE_DEFAULT);
  }

  DEFAULT {
    ENFORCE(msn.UVALUE == sequence_number.UVALUE);
    pad_bit         =:= static;
    extension       =:= static;
    cc              =:= static;
    marker          =:= static;
    payload_type    =:= static;
    sequence_number =:= static;
    timestamp       =:= static;
    ssrc            =:= static;
    csrc_list       =:= static;
    ts_stride       =:= static;
    time_stride     =:= static;
    ts_scaled       =:= static;
    ts_offset       =:= static;
  }

  COMPRESSED rtp_static {
    ssrc            =:= irregular(32)  [ 32 ];
  }

  COMPRESSED rtp_dynamic {
    reserved        =:= compressed_value(1, 0)       [  1 ];
    reorder_ratio   =:= irregular(2)                 [  2 ];
    list_present    =:= irregular(1)                 [  1 ];
    tss_indicator   =:= irregular(1)                 [  1 ];
    tis_indicator   =:= irregular(1)                 [  1 ];
    pad_bit         =:= irregular(1)                 [  1 ];
    extension       =:= irregular(1)                 [  1 ];
    marker          =:= irregular(1)                 [  1 ];
    payload_type    =:= irregular(7)                 [  7 ];
    sequence_number =:= irregular(16)                [ 16 ];
    timestamp       =:= irregular(32)                [ 32 ];
    ts_stride       =:= sdvl_or_default(tss_indicator.CVALUE,
      TS_STRIDE_DEFAULT)                             [ VARIABLE ];
```

```
    time_stride     =:= sdvl_or_default(tis_indicator.CVALUE,
      TIME_STRIDE_DEFAULT)                          [ VARIABLE ];
    csrc_list   =:= csrc_list_dynchain(list_present.CVALUE,
      cc.UVALUE)                                    [ VARIABLE ];
  }

  COMPRESSED rtp_irregular {
  }
}

//////////////////////////////////////////
// UDP-Lite Header
//////////////////////////////////////////

checksum_coverage_dynchain(behavior)
{
  UNCOMPRESSED {
    checksum_coverage [ 16 ];
  }

  COMPRESSED inferred_coverage {
    ENFORCE(behavior == UDP_LITE_COVERAGE_INFERRED);
    checksum_coverage =:= inferred_udp_length [  0 ];
  }

  COMPRESSED static_coverage {
    ENFORCE(behavior == UDP_LITE_COVERAGE_STATIC);
    checksum_coverage =:= irregular(16)       [ 16 ];
  }

  COMPRESSED irregular_coverage {
    ENFORCE(behavior == UDP_LITE_COVERAGE_IRREGULAR);
    checksum_coverage =:= irregular(16)       [ 16 ];
  }
}

checksum_coverage_irregular(behavior)
{
  UNCOMPRESSED {
    checksum_coverage [ 16 ];
  }

  COMPRESSED inferred_coverage {
    ENFORCE(behavior == UDP_LITE_COVERAGE_INFERRED);
    checksum_coverage =:= inferred_udp_length [  0 ];
  }

  COMPRESSED static_coverage {
```

```
    ENFORCE(behavior == UDP_LITE_COVERAGE_STATIC);
    checksum_coverage =:= static              [  0 ];
  }

  COMPRESSED irregular_coverage {
    ENFORCE(behavior == UDP_LITE_COVERAGE_IRREGULAR);
    checksum_coverage =:= irregular(16)       [ 16 ];
  }
}

udp_lite(profile_value, reorder_ratio_value, coverage_behavior_value)
{
  UNCOMPRESSED {
    ENFORCE((profile_value == PROFILE_RTP_0107) ||
            (profile_value == PROFILE_UDPLITE_0108));
    src_port          [ 16 ];
    dst_port          [ 16 ];
    checksum_coverage [ 16 ];
    checksum          [ 16 ];
  }

  CONTROL {
    ENFORCE(profile == profile_value);
    ENFORCE(coverage_behavior.UVALUE == coverage_behavior_value);
    ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
  }

  DEFAULT {
    src_port          =:= static;
    dst_port          =:= static;
    coverage_behavior =:= static;
  }

  COMPRESSED udp_lite_static {
    src_port   =:= irregular(16) [ 16 ];
    dst_port   =:= irregular(16) [ 16 ];
  }

  COMPRESSED udp_lite_endpoint_dynamic {
    ENFORCE(profile_value == PROFILE_UDPLITE_0108);
    reserved =:= compressed_value(4, 0)                      [  4 ];
    coverage_behavior =:= irregular(2)                       [  2 ];
    reorder_ratio     =:= irregular(2)                       [  2 ];
    checksum_coverage =:=
      checksum_coverage_dynchain(coverage_behavior.UVALUE)   [ 16 ];
    checksum          =:= irregular(16)                      [ 16 ];
    msn               =:= irregular(16)                      [ 16 ];
  }
```

```
  COMPRESSED udp_lite_regular_dynamic {
    ENFORCE(profile_value == PROFILE_RTP_0107);
    coverage_behavior =:= irregular(2)                      [  2 ];
    reserved =:= compressed_value(6, 0)                     [  6 ];
    checksum_coverage =:=
        checksum_coverage_dynchain(coverage_behavior.UVALUE) [ 16 ];
    checksum =:= irregular(16)                              [ 16 ];
  }

  COMPRESSED udp_lite_irregular {
    checksum_coverage =:=
      checksum_coverage_irregular(coverage_behavior.UVALUE) [ 0, 16 ];
    checksum          =:= irregular(16)                     [ 16 ];
  }
}

//////////////////////////////////////////
// ESP Header
//////////////////////////////////////////

esp(profile_value, reorder_ratio_value)
{
  UNCOMPRESSED {
    ENFORCE(profile_value == PROFILE_ESP_0103);
    ENFORCE(msn.UVALUE == sequence_number.UVALUE % 65536);
    spi             [ 32 ];
    sequence_number [ 32 ];
  }

  CONTROL {
    ENFORCE(profile == profile_value);
    ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
  }

  DEFAULT {
    spi             =:= static;
    sequence_number =:= static;
  }

  COMPRESSED esp_static {
    spi =:= irregular(32)                         [ 32 ];
  }

  COMPRESSED esp_dynamic {
    sequence_number =:= irregular(32)             [ 32 ];
    reserved        =:= compressed_value(6, 0)    [  6 ];
    reorder_ratio   =:= irregular(2)              [  2 ];
  }
```

```
  COMPRESSED esp_irregular {
  }
}

////////////////////////////////////////////////
// Encoding methods used in the profiles' CO headers
////////////////////////////////////////////////

// Variable reordering offset used for MSN
msn_lsb(k)
{
  UNCOMPRESSED {
    master [ VARIABLE ];
  }

  COMPRESSED none {
    ENFORCE(reorder_ratio.UVALUE == REORDERING_NONE);
    master =:= lsb(k, 1);
  }

  COMPRESSED quarter {
    ENFORCE(reorder_ratio.UVALUE == REORDERING_QUARTER);
    master =:= lsb(k, ((2^k) / 4) - 1);
  }

  COMPRESSED half {
    ENFORCE(reorder_ratio.UVALUE == REORDERING_HALF);
    master =:= lsb(k, ((2^k) / 2) - 1);
  }

  COMPRESSED threequarters {
    ENFORCE(reorder_ratio.UVALUE == REORDERING_THREEQUARTERS);
    master =:= lsb(k, (((2^k) * 3) / 4) - 1);
  }
}

ip_id_lsb(behavior, k)
{
  UNCOMPRESSED {
    ip_id [ 16 ];
  }

  CONTROL {
    ip_id_nbo    [ 16 ];
  }

  COMPRESSED nbo {
    ENFORCE(behavior == IP_ID_BEHAVIOR_SEQUENTIAL);
```

```
    ENFORCE(ip_id_offset.UVALUE == ip_id.UVALUE - msn.UVALUE);
    ip_id_offset =:= lsb(k, ((2^k) / 4) - 1) [ k ];
  }

  COMPRESSED non_nbo {
    ENFORCE(behavior == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED);
    ENFORCE(ip_id_nbo.UVALUE ==
            (ip_id.UVALUE / 256) + (ip_id.UVALUE % 256) * 256);
    ENFORCE(ip_id_nbo.ULENGTH == 16);
    ENFORCE(ip_id_offset.UVALUE == ip_id_nbo.UVALUE - msn.UVALUE);
    ip_id_offset =:= lsb(k, ((2^k) / 4) - 1) [ k ];
  }
}

ip_id_sequential_variable(behavior, indicator)
{
  UNCOMPRESSED {
    ip_id [ 16 ];
  }

  COMPRESSED short {
    ENFORCE((behavior == IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (behavior == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    ENFORCE(indicator == 0);
    ip_id =:= ip_id_lsb(behavior, 8) [ 8 ];
  }

  COMPRESSED long {
    ENFORCE((behavior == IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (behavior == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    ENFORCE(indicator == 1);
    ENFORCE(ip_id_offset.UVALUE == ip_id.UVALUE - msn.UVALUE);
    ip_id =:= irregular(16)  [ 16 ];
  }

  COMPRESSED not_present {
    ENFORCE((behavior == IP_ID_BEHAVIOR_RANDOM) ||
            (behavior == IP_ID_BEHAVIOR_ZERO));
  }
}

dont_fragment(version)
{
  UNCOMPRESSED {
    df [ 0, 1 ];
  }

  COMPRESSED v4 {
```

```
    ENFORCE(version == 4);
    df =:= irregular(1) [ 1 ];
  }

  COMPRESSED v6 {
    ENFORCE(version == 6);
    unused =:= compressed_value(1, 0) [ 1 ];
  }
}

pt_irr_or_static(flag)
{
  UNCOMPRESSED {
    payload_type [ 7 ];
  }

  COMPRESSED not_present {
    ENFORCE(flag == 0);
    payload_type =:= static [ 0 ];
  }

  COMPRESSED present {
    ENFORCE(flag == 1);
    reserved     =:= compressed_value(1, 0) [ 1 ];
    payload_type =:= irregular(7)           [ 7 ];
  }
}

csrc_list_presence(presence, cc_value)
{
  UNCOMPRESSED {
    csrc_list;
  }

  COMPRESSED no_list {
    ENFORCE(presence == 0);
    csrc_list =:= static [ 0 ];
  }

  COMPRESSED list_present {
    ENFORCE(presence == 1);
    csrc_list =:= list_csrc(cc_value) [ VARIABLE ];
  }
}

scaled_ts_lsb(time_stride_value, k)
{
  UNCOMPRESSED {
```

```
    timestamp [ 32 ];
  }

  COMPRESSED timerbased {
    ENFORCE(time_stride_value != 0);
    timestamp =:= timer_based_lsb(time_stride_value, k,
                                  ((2^k) / 2) - 1);
  }

  COMPRESSED regular {
    ENFORCE(time_stride_value == 0);
    timestamp =:= lsb(k, ((2^k) / 4) - 1);
  }
}

// Self-describing variable length encoding with reordering offset
sdvl_sn_lsb(field_width)
{
  UNCOMPRESSED {
    field [ field_width ];
  }

  COMPRESSED lsb7 {
    discriminator =:= '0'   [ 1 ];
    field =:= msn_lsb(7)    [ 7 ];
  }

  COMPRESSED lsb14 {
    discriminator =:= '10'  [  2 ];
    field =:= msn_lsb(14)   [ 14 ];
  }

  COMPRESSED lsb21 {
    discriminator =:= '110'  [  3 ];
    field =:= msn_lsb(21)    [ 21 ];
  }

  COMPRESSED lsb28 {
    discriminator =:= '1110' [  4 ];
    field =:= msn_lsb(28)    [ 28 ];
  }

  COMPRESSED lsb32 {
    discriminator =:= '11111111'        [  8 ];
    field =:= irregular(field_width)    [ field_width ];
  }
}
```

```
// Self-describing variable length encoding
sdvl_lsb(field_width)
{
  UNCOMPRESSED {
    field [ field_width ];
  }

  COMPRESSED lsb7 {
    discriminator =:= '0'               [ 1 ];
    field =:= lsb(7, ((2^7) / 4) - 1)   [ 7 ];
  }

  COMPRESSED lsb14 {
    discriminator =:= '10'              [  2 ];
    field =:= lsb(14, ((2^14) / 4) - 1) [ 14 ];
  }

  COMPRESSED lsb21 {
    discriminator =:= '110'             [  3 ];
    field =:= lsb(21, ((2^21) / 4) - 1) [ 21 ];
  }

  COMPRESSED lsb28 {
    discriminator =:= '1110'            [  4 ];
    field =:= lsb(28, ((2^28) / 4) - 1) [ 28 ];
  }

  COMPRESSED lsb32 {
    discriminator =:= '11111111'        [  8 ];
    field =:= irregular(field_width)    [ field_width ];
  }
}

sdvl_scaled_ts_lsb(time_stride)
{
   UNCOMPRESSED {
     field [ 32 ];
   }

   COMPRESSED lsb7 {
     discriminator =:= '0'                      [  1 ];
     field =:= scaled_ts_lsb(time_stride, 7)    [  7 ];
   }

   COMPRESSED lsb14 {
     discriminator =:= '10'                     [  2 ];
     field =:= scaled_ts_lsb(time_stride, 14)   [ 14 ];
   }
```

```
   COMPRESSED lsb21 {
     discriminator =:= '110'                    [  3 ];
     field =:= scaled_ts_lsb(time_stride, 21)   [ 21 ];
   }

   COMPRESSED lsb28 {
     discriminator =:= '1110'                   [  4 ];
     field =:= scaled_ts_lsb(time_stride, 28)   [ 28 ];
   }

   COMPRESSED lsb32 {
     discriminator =:= '11111111'               [  8 ];
     field =:= irregular(32)                    [ 32 ];
   }
}

variable_scaled_timestamp(tss_flag, tsc_flag, ts_stride, time_stride)
{
  UNCOMPRESSED {
    scaled_value [ 32 ];
  }

  COMPRESSED present {
    ENFORCE((tss_flag == 0) && (tsc_flag == 1));
    ENFORCE(ts_stride != 0);
    scaled_value =:= sdvl_scaled_ts_lsb(time_stride) [ VARIABLE ];
  }

  COMPRESSED not_present {
    ENFORCE(((tss_flag == 1) && (tsc_flag == 0)) ||
            ((tss_flag == 0) && (tsc_flag == 0)));
  }
}

variable_unscaled_timestamp(tss_flag, tsc_flag)
{
  UNCOMPRESSED {
    timestamp [ 32 ];
  }

  COMPRESSED present {
    ENFORCE(((tss_flag == 1) && (tsc_flag == 0)) ||
            ((tss_flag == 0) && (tsc_flag == 0)));
    timestamp =:= sdvl_lsb(32);
  }

  COMPRESSED not_present {
    ENFORCE((tss_flag == 0) && (tsc_flag == 1));
```

```
  }
}

profile_1_7_flags1_enc(flag, ip_version)
{
  UNCOMPRESSED {
    ip_outer_indicator  [ 1 ];
    ttl_hopl_indicator  [ 1 ];
    tos_tc_indicator    [ 1 ];
    df                  [ 0, 1 ];
    ip_id_behavior      [ 2 ];
    reorder_ratio       [ 2 ];
  }

  COMPRESSED not_present {
    ENFORCE(flag == 0);
    ENFORCE(ip_outer_indicator.CVALUE == 0);
    ENFORCE(ttl_hopl_indicator.CVALUE == 0);
    ENFORCE(tos_tc_indicator.CVALUE == 0);
    df                   =:= static;
    ip_id_behavior       =:= static;
    reorder_ratio        =:= static;
  }

  COMPRESSED present {
    ENFORCE(flag == 1);
    ip_outer_indicator  =:= irregular(1)                [ 1 ];
    ttl_hopl_indicator  =:= irregular(1)                [ 1 ];
    tos_tc_indicator    =:= irregular(1)                [ 1 ];
    df                  =:= dont_fragment(ip_version)   [ 1 ];
    ip_id_behavior      =:= irregular(2)                [ 2 ];
    reorder_ratio       =:= irregular(2)                [ 2 ];
  }
}

profile_1_flags2_enc(flag)
{
  UNCOMPRESSED {
    list_indicator        [ 1 ];
    pt_indicator          [ 1 ];
    time_stride_indicator [ 1 ];
    pad_bit               [ 1 ];
    extension             [ 1 ];
  }

  COMPRESSED not_present{
    ENFORCE(flag == 0);
    ENFORCE(list_indicator.UVALUE == 0);
```

```
    ENFORCE(pt_indicator.UVALUE == 0);
    ENFORCE(time_stride_indicator.UVALUE == 0);
    pad_bit      =:= static;
    extension    =:= static;
  }

  COMPRESSED present {
    ENFORCE(flag == 1);
    list_indicator =:= irregular(1)                 [ 1 ];
    pt_indicator   =:= irregular(1)                 [ 1 ];
    time_stride_indicator =:= irregular(1)          [ 1 ];
    pad_bit        =:= irregular(1)                 [ 1 ];
    extension      =:= irregular(1)                 [ 1 ];
    reserved       =:= compressed_value(3, 0)       [ 3 ];
  }
}

profile_2_3_4_flags_enc(flag, ip_version)
{
  UNCOMPRESSED {
    ip_outer_indicator [ 1 ];
    df                 [ 0, 1 ];
    ip_id_behavior     [ 2 ];
  }

  COMPRESSED not_present {
    ENFORCE(flag == 0);
    ENFORCE(ip_outer_indicator.CVALUE == 0);
    df                 =:= static;
    ip_id_behavior     =:= static;
  }

  COMPRESSED present {
    ENFORCE(flag == 1);
    ip_outer_indicator =:= irregular(1)             [ 1 ];
    df                 =:= dont_fragment(ip_version) [ 1 ];
    ip_id_behavior     =:= irregular(2)             [ 2 ];
    reserved           =:= compressed_value(4, 0)   [ 4 ];
  }
}

profile_8_flags_enc(flag, ip_version)
{
  UNCOMPRESSED {
    ip_outer_indicator  [ 1 ];
    df                  [ 0, 1 ];
    ip_id_behavior      [ 2 ];
    coverage_behavior   [ 2 ];
```

```
  }

  COMPRESSED not_present {
    ENFORCE(flag == 0);
    ENFORCE(ip_outer_indicator.CVALUE == 0);
    df                  =:= static;
    ip_id_behavior      =:= static;
    coverage_behavior   =:= static;
  }

  COMPRESSED present {
    ENFORCE(flag == 1);
    reserved            =:= compressed_value(2, 0)      [ 2 ];
    ip_outer_indicator  =:= irregular(1)                [ 1 ];
    df                  =:= dont_fragment(ip_version)   [ 1 ];
    ip_id_behavior      =:= irregular(2)                [ 2 ];
    coverage_behavior   =:= irregular(2)                [ 2 ];
  }
}

profile_7_flags2_enc(flag)
{
  UNCOMPRESSED {
    list_indicator          [ 1 ];
    pt_indicator            [ 1 ];
    time_stride_indicator   [ 1 ];
    pad_bit                 [ 1 ];
    extension               [ 1 ];
    coverage_behavior       [ 2 ];
  }

  COMPRESSED not_present{
    ENFORCE(flag == 0);
    ENFORCE(list_indicator.CVALUE == 0);
    ENFORCE(pt_indicator.CVALUE == 0);
    ENFORCE(time_stride_indicator.CVALUE == 0);
    pad_bit             =:= static;
    extension           =:= static;
    coverage_behavior   =:= static;
  }

  COMPRESSED present {
    ENFORCE(flag == 1);
    reserved       =:= compressed_value(1, 0)      [ 1 ];
    list_indicator =:= irregular(1)                [ 1 ];
    pt_indicator   =:= irregular(1)                [ 1 ];
    time_stride_indicator =:= irregular(1)         [ 1 ];
    pad_bit        =:= irregular(1)                [ 1 ];
```

```
    extension      =:= irregular(1)               [ 1 ];
    coverage_behavior =:= irregular(2)            [ 2 ];
  }
}

//////////////////////////////////////////
// RTP profile
//////////////////////////////////////////

rtp_baseheader(profile_value, ts_stride_value, time_stride_value,
               outer_ip_flag, ip_id_behavior_value,
               reorder_ratio_value)
{
  UNCOMPRESSED v4 {
    ENFORCE(msn.UVALUE == sequence_number.UVALUE);
    outer_headers  =:= baseheader_outer_headers        [ VARIABLE ];
    ip_version     =:= uncompressed_value(4, 4)        [  4 ];
    header_length  =:= uncompressed_value(4, 5)        [  4 ];
    tos_tc                                             [  8 ];
    length         =:= inferred_ip_v4_length           [ 16 ];
    ip_id                                              [ 16 ];
    rf             =:= uncompressed_value(1, 0)        [  1 ];
    df                                                 [  1 ];
    mf             =:= uncompressed_value(1, 0)        [  1 ];
    frag_offset    =:= uncompressed_value(13, 0)       [ 13 ];
    ttl_hopl                                           [  8 ];
    next_header                                        [  8 ];
    ip_checksum =:= inferred_ip_v4_header_checksum     [ 16 ];
    src_addr                                           [ 32 ];
    dest_addr                                          [ 32 ];
    extension_headers =:= baseheader_extension_headers [ VARIABLE ];
    src_port                                           [ 16 ];
    dst_port                                           [ 16 ];
    udp_length  =:= inferred_udp_length                [ 16 ];
    udp_checksum                                       [ 16 ];
    rtp_version =:= uncompressed_value(2, 2)           [  2 ];
    pad_bit                                            [  1 ];
    extension                                          [  1 ];
    cc                                                 [  4 ];
    marker                                             [  1 ];
    payload_type                                       [  7 ];
    sequence_number                                    [ 16 ];
    timestamp                                          [ 32 ];
    ssrc                                               [ 32 ];
    csrc_list                                          [ VARIABLE ];
  }

  UNCOMPRESSED v6 {
```

```
    ENFORCE(ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_RANDOM);
    ENFORCE(msn.UVALUE == sequence_number.UVALUE);
    outer_headers  =:= baseheader_outer_headers        [ VARIABLE ];
    ip_version     =:= uncompressed_value(4, 6)        [   4 ];
    tos_tc                                             [   8 ];
    flow_label                                         [  20 ];
    payload_length =:= inferred_ip_v6_length           [  16 ];
    next_header                                        [   8 ];
    ttl_hopl                                           [   8 ];
    src_addr                                           [ 128 ];
    dest_addr                                          [ 128 ];
    extension_headers =:= baseheader_extension_headers [ VARIABLE ];
    src_port                                           [  16 ];
    dst_port                                           [  16 ];
    udp_length     =:= inferred_udp_length             [  16 ];
    udp_checksum                                       [  16 ];
    rtp_version    =:= uncompressed_value(2, 2)        [   2 ];
    pad_bit                                            [   1 ];
    extension                                          [   1 ];
    cc                                                 [   4 ];
    marker                                             [   1 ];
    payload_type                                       [   7 ];
    sequence_number                                    [  16 ];
    timestamp                                          [  32 ];
    ssrc                                               [  32 ];
    csrc_list                                          [ VARIABLE ];
    df    =:= uncompressed_value(0,0)                  [   0 ];
    ip_id =:= uncompressed_value(0,0)                  [   0 ];
  }

  CONTROL {
    ENFORCE(profile_value == PROFILE_RTP_0101);
    ENFORCE(profile == profile_value);
    ENFORCE(time_stride.UVALUE == time_stride_value);
    ENFORCE(ts_stride.UVALUE == ts_stride_value);
    ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
    ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
    dummy_field =:= field_scaling(ts_stride.UVALUE,
      ts_scaled.UVALUE, timestamp.UVALUE, ts_offset.UVALUE) [ 0 ];
  }

  INITIAL {
    ts_stride     =:= uncompressed_value(32, TS_STRIDE_DEFAULT);
    time_stride   =:= uncompressed_value(32, TIME_STRIDE_DEFAULT);
  }

  DEFAULT {
    ENFORCE(outer_ip_flag == 0);
```

```
    tos_tc          =:= static;
    dest_addr       =:= static;
    ttl_hopl        =:= static;
    src_addr        =:= static;
    df              =:= static;
    flow_label      =:= static;
    next_header     =:= static;
    src_port        =:= static;
    dst_port        =:= static;
    pad_bit         =:= static;
    extension       =:= static;
    cc              =:= static;
    // When marker not present in packets, it is assumed 0
    marker          =:= uncompressed_value(1, 0);
    payload_type    =:= static;
    sequence_number =:= static;
    timestamp       =:= static;
    ssrc            =:= static;
    csrc_list       =:= static;
    ts_stride       =:= static;
    time_stride     =:= static;
    ts_scaled       =:= static;
    ts_offset       =:= static;
    reorder_ratio   =:= static;
    ip_id_behavior_innermost =:= static;
  }

  // Replacement for UOR-2-ext3
  COMPRESSED co_common {
    ENFORCE(outer_ip_flag == outer_ip_indicator.CVALUE);
    discriminator        =:= '11111010'                    [ 8 ];
    marker               =:= irregular(1)                  [ 1 ];
    header_crc   =:= crc7(THIS.UVALUE, THIS.ULENGTH)       [ 7 ];
    flags1_indicator     =:= irregular(1)                  [ 1 ];
    flags2_indicator     =:= irregular(1)                  [ 1 ];
    tsc_indicator        =:= irregular(1)                  [ 1 ];
    tss_indicator        =:= irregular(1)                  [ 1 ];
    ip_id_indicator      =:= irregular(1)                  [ 1 ];
    control_crc3         =:= control_crc3_encoding         [ 3 ];

    outer_ip_indicator : ttl_hopl_indicator :
      tos_tc_indicator : df : ip_id_behavior_innermost : reorder_ratio
      =:= profile_1_7_flags1_enc(flags1_indicator.CVALUE,
        ip_version.UVALUE)                                 [ 0, 8 ];
    list_indicator : pt_indicator : tis_indicator : pad_bit :
      extension =:= profile_1_flags2_enc(
        flags2_indicator.CVALUE)                           [ 0, 8 ];
    tos_tc =:= static_or_irreg(tos_tc_indicator.CVALUE, 8) [ 0, 8 ];
```

```
    ttl_hopl =:= static_or_irreg(ttl_hopl_indicator.CVALUE,
      ttl_hopl.ULENGTH)                                    [ 0, 8 ];
    payload_type =:= pt_irr_or_static(pt_indicator)        [ 0, 8 ];
    sequence_number =:=
      sdvl_sn_lsb(sequence_number.ULENGTH)                [ VARIABLE ];
    ip_id =:= ip_id_sequential_variable(
      ip_id_behavior_innermost.UVALUE,
      ip_id_indicator.CVALUE) [ 0, 8, 16 ];
    ts_scaled =:= variable_scaled_timestamp(tss_indicator.CVALUE,
      tsc_indicator.CVALUE, ts_stride.UVALUE,
      time_stride.UVALUE)                                  [ VARIABLE ];
    timestamp =:= variable_unscaled_timestamp(tss_indicator.CVALUE,
      tsc_indicator.CVALUE)                                [ VARIABLE ];
    ts_stride =:= sdvl_or_static(tss_indicator.CVALUE)     [ VARIABLE ];
    time_stride =:= sdvl_or_static(tis_indicator.CVALUE)   [ VARIABLE ];
    csrc_list =:= csrc_list_presence(list_indicator.CVALUE,
      cc.UVALUE)                                           [ VARIABLE ];
  }

  // UO-0
  COMPRESSED pt_0_crc3 {
    discriminator =:= '0'                              [ 1 ];
    msn           =:= msn_lsb(4)                       [ 4 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)  [ 3 ];
    timestamp     =:= inferred_scaled_field            [ 0 ];
    ip_id         =:= inferred_sequential_ip_id        [ 0 ];
  }

  // New format, Type 0 with strong CRC and more SN bits
  COMPRESSED pt_0_crc7 {
    discriminator =:= '1000'                           [ 4 ];
    msn           =:= msn_lsb(5)                       [ 5 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)  [ 7 ];
    timestamp     =:= inferred_scaled_field            [ 0 ];
    ip_id         =:= inferred_sequential_ip_id        [ 0 ];
  }

  // UO-1 replacement
  COMPRESSED pt_1_rnd {
    ENFORCE(ts_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_RANDOM) ||
            (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_ZERO));
    discriminator =:= '101'                            [ 3 ];
    marker        =:= irregular(1)                     [ 1 ];
    msn           =:= msn_lsb(4)                       [ 4 ];
    ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 5) [ 5 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)  [ 3 ];
```

```
  }

  // UO-1-ID replacement
  COMPRESSED pt_1_seq_id {
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '1001'                                  [ 4 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 4) [ 4 ];
    msn           =:= msn_lsb(5)                              [ 5 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)         [ 3 ];
    timestamp     =:= inferred_scaled_field                   [ 0 ];
  }

  // UO-1-TS replacement
  COMPRESSED pt_1_seq_ts {
    ENFORCE(ts_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '101'                                 [ 3 ];
    marker        =:= irregular(1)                          [ 1 ];
    msn           =:= msn_lsb(4)                            [ 4 ];
    ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 5) [ 5 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)       [ 3 ];
    ip_id         =:= inferred_sequential_ip_id       [ 0 ];
  }

  // UOR-2 replacement
  COMPRESSED pt_2_rnd {
    ENFORCE(ts_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_RANDOM) ||
            (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_ZERO));
    discriminator =:= '110'                                 [ 3 ];
    msn           =:= msn_lsb(7)                            [ 7 ];
    ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 6) [ 6 ];
    marker        =:= irregular(1)                          [ 1 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)       [ 7 ];
  }

  // UOR-2-ID replacement
  COMPRESSED pt_2_seq_id {
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
```

```
            IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '11000'                                  [ 5 ];
    msn           =:= msn_lsb(7)                               [ 7 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 5)    [ 5 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)          [ 7 ];
    timestamp     =:= inferred_scaled_field                    [ 0 ];
  }

  // UOR-2-ID-ext1 replacement (both TS and IP-ID)
  COMPRESSED pt_2_seq_both {
    ENFORCE(ts_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '11001'                                  [ 5 ];
    msn           =:= msn_lsb(7)                               [ 7 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 5)    [ 5 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)          [ 7 ];
    ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 7)     [ 7 ];
    marker        =:= irregular(1)                             [ 1 ];
  }

  // UOR-2-TS replacement
  COMPRESSED pt_2_seq_ts {
    ENFORCE(ts_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '1101'                                   [ 4 ];
    msn           =:= msn_lsb(7)                               [ 7 ];
    ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 5)     [ 5 ];
    marker        =:= irregular(1)                             [ 1 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)          [ 7 ];
    ip_id         =:= inferred_sequential_ip_id                [ 0 ];
  }
}

//////////////////////////////////////////
// UDP profile
//////////////////////////////////////////

udp_baseheader(profile_value, outer_ip_flag, ip_id_behavior_value,
               reorder_ratio_value)
{
  UNCOMPRESSED v4 {
    outer_headers  =:= baseheader_outer_headers        [ VARIABLE ];
```

```
    ip_version     =:= uncompressed_value(4, 4)        [  4 ];
    header_length  =:= uncompressed_value(4, 5)        [  4 ];
    tos_tc                                             [  8 ];
    length         =:= inferred_ip_v4_length           [ 16 ];
    ip_id                                              [ 16 ];
    rf             =:= uncompressed_value(1, 0)        [  1 ];
    df                                                 [  1 ];
    mf             =:= uncompressed_value(1, 0)        [  1 ];
    frag_offset    =:= uncompressed_value(13, 0)       [ 13 ];
    ttl_hopl                                           [  8 ];
    next_header                                        [  8 ];
    ip_checksum =:= inferred_ip_v4_header_checksum     [ 16 ];
    src_addr                                           [ 32 ];
    dest_addr                                          [ 32 ];
    extension_headers =:= baseheader_extension_headers [ VARIABLE ];
    src_port                                           [ 16 ];
    dst_port                                           [ 16 ];
    udp_length     =:= inferred_udp_length             [ 16 ];
    udp_checksum                                       [ 16 ];
  }

  UNCOMPRESSED v6 {
    ENFORCE(ip_id_behavior.UVALUE == IP_ID_BEHAVIOR_RANDOM);
    outer_headers  =:= baseheader_outer_headers        [ VARIABLE ];
    ip_version     =:= uncompressed_value(4, 6)        [  4 ];
    tos_tc                                             [  8 ];
    flow_label                                         [ 20 ];
    payload_length =:= inferred_ip_v6_length           [ 16 ];
    next_header                                        [  8 ];
    ttl_hopl                                           [  8 ];
    src_addr                                           [ 128 ];
    dest_addr                                          [ 128 ];
    extension_headers =:= baseheader_extension_headers [ VARIABLE ];
    src_port                                           [ 16 ];
    dst_port                                           [ 16 ];
    udp_length     =:= inferred_udp_length             [ 16 ];
    udp_checksum                                       [ 16 ];
    df    =:= uncompressed_value(0,0)                  [  0 ];
    ip_id =:= uncompressed_value(0,0)                  [  0 ];
  }

  CONTROL {
    ENFORCE(profile_value == PROFILE_UDP_0102);
    ENFORCE(profile == profile_value);
    ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
    ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
  }
```

```
  DEFAULT {
    ENFORCE(outer_ip_flag == 0);
    tos_tc         =:= static;
    dest_addr      =:= static;
    ip_version     =:= static;
    ttl_hopl       =:= static;
    src_addr       =:= static;
    df             =:= static;
    flow_label     =:= static;
    next_header    =:= static;
    src_port       =:= static;
    dst_port       =:= static;
    reorder_ratio  =:= static;
    ip_id_behavior_innermost =:= static;
  }

  // Replacement for UOR-2-ext3
  COMPRESSED co_common {
    ENFORCE(outer_ip_flag == outer_ip_indicator.CVALUE);
    discriminator        =:= '11111010'                    [ 8 ];
    ip_id_indicator      =:= irregular(1)                  [ 1 ];
    header_crc   =:= crc7(THIS.UVALUE, THIS.ULENGTH)       [ 7 ];
    flags_indicator      =:= irregular(1)                  [ 1 ];
    ttl_hopl_indicator   =:= irregular(1)                  [ 1 ];
    tos_tc_indicator     =:= irregular(1)                  [ 1 ];
    reorder_ratio        =:= irregular(2)                  [ 2 ];
    control_crc3         =:= control_crc3_encoding         [ 3 ];
    outer_ip_indicator : df : ip_id_behavior_innermost =:=
      profile_2_3_4_flags_enc(
      flags_indicator.CVALUE, ip_version.UVALUE)           [ 0, 8 ];
    tos_tc =:= static_or_irreg(tos_tc_indicator.CVALUE, 8) [ 0, 8 ];
    ttl_hopl =:= static_or_irreg(ttl_hopl_indicator.CVALUE,
      ttl_hopl.ULENGTH)                                    [ 0, 8 ];
    msn                  =:= msn_lsb(8)                    [ 8 ];
    ip_id =:= ip_id_sequential_variable(ip_id_behavior_innermost.UVALUE,
      ip_id_indicator.CVALUE)                             [ 0, 8, 16 ];
  }

  // UO-0
  COMPRESSED pt_0_crc3 {
    discriminator =:= '0'                                 [ 1 ];
    msn           =:= msn_lsb(4)                          [ 4 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)     [ 3 ];
    ip_id         =:= inferred_sequential_ip_id           [ 0 ];
  }

  // New format, Type 0 with strong CRC and more SN bits
  COMPRESSED pt_0_crc7 {
```

```
    discriminator =:= '100'                              [ 3 ];
    msn           =:= msn_lsb(6)                         [ 6 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)    [ 7 ];
    ip_id         =:= inferred_sequential_ip_id          [ 0 ];
  }

  // UO-1-ID replacement (PT-1 only used for sequential)
  COMPRESSED pt_1_seq_id {
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '101'                                 [ 3 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)       [ 3 ];
    msn           =:= msn_lsb(6)                            [ 6 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 4) [ 4 ];
  }

  // UOR-2-ID replacement
  COMPRESSED pt_2_seq_id {
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '110'                                 [ 3 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 6) [ 6 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)       [ 7 ];
    msn           =:= msn_lsb(8)                            [ 8 ];
  }
}

//////////////////////////////////////////
// ESP profile
//////////////////////////////////////////

esp_baseheader(profile_value, outer_ip_flag, ip_id_behavior_value,
               reorder_ratio_value)
{
  UNCOMPRESSED v4 {
    ENFORCE(msn.UVALUE == sequence_number.UVALUE % 65536);
    outer_headers  =:= baseheader_outer_headers        [ VARIABLE ];
    ip_version     =:= uncompressed_value(4, 4)        [  4 ];
    header_length  =:= uncompressed_value(4, 5)        [  4 ];
    tos_tc                                             [  8 ];
    length         =:= inferred_ip_v4_length           [ 16 ];
    ip_id                                              [ 16 ];
    rf             =:= uncompressed_value(1, 0)        [  1 ];
    df                                                 [  1 ];
```

```
    mf             =:= uncompressed_value(1, 0)        [  1 ];
    frag_offset    =:= uncompressed_value(13, 0)       [ 13 ];
    ttl_hopl                                           [  8 ];
    next_header                                        [  8 ];
    ip_checksum =:= inferred_ip_v4_header_checksum     [ 16 ];
    src_addr                                           [ 32 ];
    dest_addr                                          [ 32 ];
    extension_headers =:= baseheader_extension_headers [ VARIABLE ];
    spi                                                [ 32 ];
    sequence_number                                    [ 32 ];
  }

  UNCOMPRESSED v6 {
    ENFORCE(msn.UVALUE == (sequence_number.UVALUE % 65536));
    ENFORCE(ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_RANDOM);
    outer_headers  =:= baseheader_outer_headers        [ VARIABLE ];
    ip_version     =:= uncompressed_value(4, 6)        [   4 ];
    tos_tc                                             [   8 ];
    flow_label                                         [  20 ];
    payload_length =:= inferred_ip_v6_length           [  16 ];
    next_header                                        [   8 ];
    ttl_hopl                                           [   8 ];
    src_addr                                           [ 128 ];
    dest_addr                                          [ 128 ];
    extension_headers =:= baseheader_extension_headers [ VARIABLE ];
    spi                                                [  32 ];
    sequence_number                                    [  32 ];
    df    =:= uncompressed_value(0,0)                  [   0 ];
    ip_id =:= uncompressed_value(0,0)                  [   0 ];
  }

  CONTROL {
    ENFORCE(profile_value == PROFILE_ESP_0103);
    ENFORCE(profile == profile_value);
    ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
    ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
  }

  DEFAULT {
    ENFORCE(outer_ip_flag == 0);
    tos_tc          =:= static;
    dest_addr       =:= static;
    ttl_hopl        =:= static;
    src_addr        =:= static;
    df              =:= static;
    flow_label      =:= static;
    next_header     =:= static;
    spi             =:= static;
```

```
    sequence_number =:= static;
    reorder_ratio   =:= static;
    ip_id_behavior_innermost =:= static;
  }

  // Replacement for UOR-2-ext3
  COMPRESSED co_common {
    ENFORCE(outer_ip_flag == outer_ip_indicator.CVALUE);
    discriminator        =:= '11111010'                    [ 8 ];
    ip_id_indicator      =:= irregular(1)                  [ 1 ];
    header_crc   =:= crc7(THIS.UVALUE, THIS.ULENGTH)       [ 7 ];
    flags_indicator      =:= irregular(1)                  [ 1 ];
    ttl_hopl_indicator   =:= irregular(1)                  [ 1 ];
    tos_tc_indicator     =:= irregular(1)                  [ 1 ];
    reorder_ratio        =:= irregular(2)                  [ 2 ];
    control_crc3         =:= control_crc3_encoding         [ 3 ];

    outer_ip_indicator : df : ip_id_behavior_innermost =:=
      profile_2_3_4_flags_enc(
      flags_indicator.CVALUE, ip_version.UVALUE)           [ 0, 8 ];
    tos_tc =:= static_or_irreg(tos_tc_indicator.CVALUE, 8) [ 0, 8 ];
    ttl_hopl =:= static_or_irreg(ttl_hopl_indicator.CVALUE,
      ttl_hopl.ULENGTH)                                    [ 0, 8 ];
    sequence_number =:=
      sdvl_sn_lsb(sequence_number.ULENGTH)             [ VARIABLE ];
    ip_id =:= ip_id_sequential_variable(ip_id_behavior_innermost.UVALUE,
      ip_id_indicator.CVALUE)                          [ 0, 8, 16 ];
  }

  // Sequence number sent instead of MSN due to field length
  // UO-0
  COMPRESSED pt_0_crc3 {
    discriminator   =:= '0'                             [ 1 ];
    sequence_number =:= msn_lsb(4)                      [ 4 ];
    header_crc      =:= crc3(THIS.UVALUE, THIS.ULENGTH) [ 3 ];
    ip_id           =:= inferred_sequential_ip_id       [ 0 ];
  }

  // New format, Type 0 with strong CRC and more SN bits
  COMPRESSED pt_0_crc7 {
    discriminator   =:= '100'                           [ 3 ];
    sequence_number =:= msn_lsb(6)                      [ 6 ];
    header_crc      =:= crc7(THIS.UVALUE, THIS.ULENGTH) [ 7 ];
    ip_id           =:= inferred_sequential_ip_id       [ 0 ];
  }

  // UO-1-ID replacement (PT-1 only used for sequential)
  COMPRESSED pt_1_seq_id {
```

```
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator   =:= '101'                                  [ 3 ];
    header_crc      =:= crc3(THIS.UVALUE, THIS.ULENGTH)        [ 3 ];
    sequence_number =:= msn_lsb(6)                             [ 6 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 4)    [ 4 ];
  }

  // UOR-2-ID replacement
  COMPRESSED pt_2_seq_id {
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator   =:= '110'                                  [ 3 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 6)    [ 6 ];
    header_crc      =:= crc7(THIS.UVALUE, THIS.ULENGTH)        [ 7 ];
    sequence_number =:= msn_lsb(8)                             [ 8 ];
  }
}

////////////////////////////////////////
// IP-only profile
////////////////////////////////////////

iponly_baseheader(profile_value, outer_ip_flag, ip_id_behavior_value,
                  reorder_ratio_value)
{
  UNCOMPRESSED v4 {
    outer_headers  =:= baseheader_outer_headers        [ VARIABLE ];
    ip_version     =:= uncompressed_value(4, 4)        [  4 ];
    header_length  =:= uncompressed_value(4, 5)        [  4 ];
    tos_tc                                             [  8 ];
    length         =:= inferred_ip_v4_length           [ 16 ];
    ip_id                                              [ 16 ];
    rf             =:= uncompressed_value(1, 0)        [  1 ];
    df                                                 [  1 ];
    mf             =:= uncompressed_value(1, 0)        [  1 ];
    frag_offset    =:= uncompressed_value(13, 0)       [ 13 ];
    ttl_hopl                                           [  8 ];
    next_header                                        [  8 ];
    ip_checksum =:= inferred_ip_v4_header_checksum     [ 16 ];
    src_addr                                           [ 32 ];
    dest_addr                                          [ 32 ];
    extension_headers =:= baseheader_extension_headers [ VARIABLE ];
  }
```

```
  UNCOMPRESSED v6 {
    ENFORCE(ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_RANDOM);
    outer_headers     =:= baseheader_outer_headers     [ VARIABLE ];
    ip_version        =:= uncompressed_value(4, 6)     [   4 ];
    tos_tc                                             [   8 ];
    flow_label                                         [  20 ];
    payload_length    =:= inferred_ip_v6_length        [  16 ];
    next_header                                        [   8 ];
    ttl_hopl                                           [   8 ];
    src_addr                                           [ 128 ];
    dest_addr                                          [ 128 ];
    extension_headers =:= baseheader_extension_headers [ VARIABLE ];
    df    =:= uncompressed_value(0,0)                  [   0 ];
    ip_id =:= uncompressed_value(0,0)                  [   0 ];
  }

  CONTROL {
    ENFORCE(profile_value == PROFILE_IP_0104);
    ENFORCE(profile == profile_value);
    ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
    ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
  }

  DEFAULT {
    ENFORCE(outer_ip_flag == 0);
    tos_tc         =:= static;
    dest_addr      =:= static;
    ttl_hopl       =:= static;
    src_addr       =:= static;
    df             =:= static;
    flow_label     =:= static;
    next_header    =:= static;
    reorder_ratio  =:= static;
    ip_id_behavior_innermost =:= static;
  }

  // Replacement for UOR-2-ext3
  COMPRESSED co_common {
    ENFORCE(outer_ip_flag == outer_ip_indicator.CVALUE);
    discriminator        =:= '11111010'                    [ 8 ];
    ip_id_indicator      =:= irregular(1)                  [ 1 ];
    header_crc   =:= crc7(THIS.UVALUE, THIS.ULENGTH)       [ 7 ];
    flags_indicator      =:= irregular(1)                  [ 1 ];
    ttl_hopl_indicator   =:= irregular(1)                  [ 1 ];
    tos_tc_indicator     =:= irregular(1)                  [ 1 ];
    reorder_ratio        =:= irregular(2)                  [ 2 ];
    control_crc3         =:= control_crc3_encoding         [ 3 ];
    outer_ip_indicator : df : ip_id_behavior_innermost =:=
```

```
      profile_2_3_4_flags_enc(
      flags_indicator.CVALUE, ip_version.UVALUE)          [ 0, 8 ];
    tos_tc =:= static_or_irreg(tos_tc_indicator.CVALUE, 8) [ 0, 8 ];
    ttl_hopl =:= static_or_irreg(ttl_hopl_indicator.CVALUE,
      ttl_hopl.ULENGTH)                                   [ 0, 8 ];
    msn                =:= msn_lsb(8)                     [ 8 ];
    ip_id =:= ip_id_sequential_variable(ip_id_behavior_innermost.UVALUE,
      ip_id_indicator.CVALUE)                          [ 0, 8, 16 ];
  }

  // UO-0
  COMPRESSED pt_0_crc3 {
    discriminator =:= '0'                             [ 1 ];
    msn           =:= msn_lsb(4)                      [ 4 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH) [ 3 ];
    ip_id         =:= inferred_sequential_ip_id       [ 0 ];
  }

  // New format, Type 0 with strong CRC and more SN bits
  COMPRESSED pt_0_crc7 {
    discriminator =:= '100'                           [ 3 ];
    msn           =:= msn_lsb(6)                      [ 6 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH) [ 7 ];
    ip_id         =:= inferred_sequential_ip_id       [ 0 ];
  }

  // UO-1-ID replacement (PT-1 only used for sequential)
  COMPRESSED pt_1_seq_id {
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '101'                             [ 3 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)   [ 3 ];
    msn           =:= msn_lsb(6)                        [ 6 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 4) [ 4 ];
  }

  // UOR-2-ID replacement
  COMPRESSED pt_2_seq_id {
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '110'                             [ 3 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 6) [ 6 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)   [ 7 ];
    msn           =:= msn_lsb(8)                        [ 8 ];
```

```
  }
}

////////////////////////////////////////
// UDP-lite/RTP profile
////////////////////////////////////////

udplite_rtp_baseheader(profile_value, ts_stride_value,
                       time_stride_value, outer_ip_flag,
                       ip_id_behavior_value, reorder_ratio_value,
                       coverage_behavior_value)
{
  UNCOMPRESSED v4 {
    ENFORCE(msn.UVALUE == sequence_number.UVALUE);
    outer_headers  =:= baseheader_outer_headers        [ VARIABLE ];
    ip_version     =:= uncompressed_value(4, 4)        [  4 ];
    header_length  =:= uncompressed_value(4, 5)        [  4 ];
    tos_tc                                             [  8 ];
    length         =:= inferred_ip_v4_length           [ 16 ];
    ip_id                                              [ 16 ];
    rf             =:= uncompressed_value(1, 0)        [  1 ];
    df                                                 [  1 ];
    mf             =:= uncompressed_value(1, 0)        [  1 ];
    frag_offset    =:= uncompressed_value(13, 0)       [ 13 ];
    ttl_hopl                                           [  8 ];
    next_header                                        [  8 ];
    ip_checksum =:= inferred_ip_v4_header_checksum     [ 16 ];
    src_addr                                           [ 32 ];
    dest_addr                                          [ 32 ];
    extension_headers =:= baseheader_extension_headers [ VARIABLE ];
    src_port                                           [ 16 ];
    dst_port                                           [ 16 ];
    checksum_coverage                                  [ 16 ];
    udp_checksum                                       [ 16 ];
    rtp_version    =:= uncompressed_value(2, 2)        [  2 ];
    pad_bit                                            [  1 ];
    extension                                          [  1 ];
    cc                                                 [  4 ];
    marker                                             [  1 ];
    payload_type                                       [  7 ];
    sequence_number                                    [ 16 ];
    timestamp                                          [ 32 ];
    ssrc                                               [ 32 ];
    csrc_list                                          [ VARIABLE ];
  }

  UNCOMPRESSED v6 {
    ENFORCE(ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_RANDOM);
```

```
    outer_headers  =:= baseheader_outer_headers        [ VARIABLE ];
    ip_version     =:= uncompressed_value(4, 6)        [   4 ];
    tos_tc                                             [   8 ];
    flow_label                                         [  20 ];
    payload_length =:= inferred_ip_v6_length           [  16 ];
    next_header                                        [   8 ];
    ttl_hopl                                           [   8 ];
    src_addr                                           [ 128 ];
    dest_addr                                          [ 128 ];
    extension_headers =:= baseheader_extension_headers [ VARIABLE ];
    src_port                                           [  16 ];
    dst_port                                           [  16 ];
    checksum_coverage                                  [  16 ];
    udp_checksum                                       [  16 ];
    rtp_version =:= uncompressed_value(2, 2)           [   2 ];
    pad_bit                                            [   1 ];
    extension                                          [   1 ];
    cc                                                 [   4 ];
    marker                                             [   1 ];
    payload_type                                       [   7 ];
    sequence_number                                    [  16 ];
    timestamp                                          [  32 ];
    ssrc                                               [  32 ];
    csrc_list                                          [ VARIABLE ];
    df    =:= uncompressed_value(0,0)                  [   0 ];
    ip_id =:= uncompressed_value(0,0)                  [   0 ];
  }

  CONTROL {
    ENFORCE(profile_value == PROFILE_RTP_0107);
    ENFORCE(profile == profile_value);
    ENFORCE(time_stride.UVALUE == time_stride_value);
    ENFORCE(ts_stride.UVALUE == ts_stride_value);
    ENFORCE(coverage_behavior.UVALUE == coverage_behavior_value);
    ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
    ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
    dummy_field =:= field_scaling(ts_stride.UVALUE,
      ts_scaled.UVALUE, timestamp.UVALUE, ts_offset.UVALUE) [ 0 ];
  }

  INITIAL {
    ts_stride     =:= uncompressed_value(32, TS_STRIDE_DEFAULT);
    time_stride   =:= uncompressed_value(32, TIME_STRIDE_DEFAULT);
  }

  DEFAULT {
    ENFORCE(outer_ip_flag == 0);
    tos_tc            =:= static;
```

```
    dest_addr         =:= static;
    ttl_hopl          =:= static;
    src_addr          =:= static;
    df                =:= static;
    flow_label        =:= static;
    next_header       =:= static;
    src_port          =:= static;
    dst_port          =:= static;
    pad_bit           =:= static;
    extension         =:= static;
    cc                =:= static;
    // When marker not present in packets, it is assumed 0
    marker            =:= uncompressed_value(1, 0);
    payload_type      =:= static;
    sequence_number   =:= static;
    timestamp         =:= static;
    ssrc              =:= static;
    csrc_list         =:= static;
    ts_stride         =:= static;
    time_stride       =:= static;
    ts_scaled         =:= static;
    ts_offset         =:= static;
    reorder_ratio     =:= static;
    ip_id_behavior_innermost =:= static;
  }

  // Replacement for UOR-2-ext3
  COMPRESSED co_common {
    ENFORCE(outer_ip_flag == outer_ip_indicator.CVALUE);
    discriminator        =:= '11111010'                    [ 8 ];
    marker               =:= irregular(1)                  [ 1 ];
    header_crc   =:= crc7(THIS.UVALUE, THIS.ULENGTH)       [ 7 ];
    flags1_indicator     =:= irregular(1)                  [ 1 ];
    flags2_indicator     =:= irregular(1)                  [ 1 ];
    tsc_indicator        =:= irregular(1)                  [ 1 ];
    tss_indicator        =:= irregular(1)                  [ 1 ];
    ip_id_indicator      =:= irregular(1)                  [ 1 ];
    control_crc3         =:= control_crc3_encoding         [ 3 ];

    outer_ip_indicator : ttl_hopl_indicator :
      tos_tc_indicator : df : ip_id_behavior_innermost : reorder_ratio
      =:= profile_1_7_flags1_enc(flags1_indicator.CVALUE,
        ip_version.UVALUE)                                 [ 0, 8 ];
    list_indicator : pt_indicator : tis_indicator : pad_bit :
      extension : coverage_behavior =:=
      profile_7_flags2_enc(flags2_indicator.CVALUE)        [ 0, 8 ];
    tos_tc =:= static_or_irreg(tos_tc_indicator.CVALUE, 8) [ 0, 8 ];
    ttl_hopl =:=
```

```
      static_or_irreg(ttl_hopl_indicator.CVALUE, 8)        [ 0, 8 ];
    payload_type =:= pt_irr_or_static(pt_indicator.CVALUE) [ 0, 8 ];
    sequence_number =:=
      sdvl_sn_lsb(sequence_number.ULENGTH)               [ VARIABLE ];
    ip_id =:= ip_id_sequential_variable(ip_id_behavior_innermost.UVALUE,
      ip_id_indicator.CVALUE)                            [ 0, 8, 16 ];
    ts_scaled =:= variable_scaled_timestamp(tss_indicator.CVALUE,
      tsc_indicator.CVALUE, ts_stride.UVALUE,
      time_stride.UVALUE)                                [ VARIABLE ];
    timestamp =:= variable_unscaled_timestamp(tss_indicator.CVALUE,
      tsc_indicator.CVALUE)                              [ VARIABLE ];
    ts_stride =:= sdvl_or_static(tss_indicator.CVALUE)   [ VARIABLE ];
    time_stride =:= sdvl_or_static(tis_indicator.CVALUE) [ VARIABLE ];
    csrc_list            =:=
        csrc_list_presence(list_indicator.CVALUE,
          cc.UVALUE)                                     [ VARIABLE ];
  }

  // UO-0
  COMPRESSED pt_0_crc3 {
    discriminator =:= '0'                              [ 1 ];
    msn           =:= msn_lsb(4)                       [ 4 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH) [ 3 ];
    timestamp     =:= inferred_scaled_field            [ 0 ];
    ip_id         =:= inferred_sequential_ip_id        [ 0 ];
  }

  // New format, Type 0 with strong CRC and more SN bits
  COMPRESSED pt_0_crc7 {
    discriminator =:= '1000'                           [ 4 ];
    msn           =:= msn_lsb(5)                       [ 5 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH) [ 7 ];
    timestamp     =:= inferred_scaled_field            [ 0 ];
    ip_id         =:= inferred_sequential_ip_id        [ 0 ];
  }

  // UO-1 replacement
  COMPRESSED pt_1_rnd {
    ENFORCE(ts_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_RANDOM) ||
            (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_ZERO));
    discriminator =:= '101'                            [ 3 ];
    marker        =:= irregular(1)                     [ 1 ];
    msn           =:= msn_lsb(4)                       [ 4 ];
    ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 5) [ 5 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)  [ 3 ];
  }
```

```
  // UO-1-ID replacement
  COMPRESSED pt_1_seq_id {
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '1001'                                  [ 4 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 4) [ 4 ];
    msn           =:= msn_lsb(5)                              [ 5 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)         [ 3 ];
    timestamp     =:= inferred_scaled_field                   [ 0 ];
  }

  // UO-1-TS replacement
  COMPRESSED pt_1_seq_ts {
    ENFORCE(ts_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '101'                                 [ 3 ];
    marker        =:= irregular(1)                          [ 1 ];
    msn           =:= msn_lsb(4)                            [ 4 ];
    ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 5) [ 5 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)       [ 3 ];
    ip_id         =:= inferred_sequential_ip_id             [ 0 ];
  }

  // UOR-2 replacement
  COMPRESSED pt_2_rnd {
    ENFORCE(ts_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_RANDOM) ||
            (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_ZERO));
    discriminator =:= '110'                                 [ 3 ];
    msn           =:= msn_lsb(7)                            [ 7 ];
    ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 6) [ 6 ];
    marker        =:= irregular(1)                          [ 1 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)       [ 7 ];
  }

  // UOR-2-ID replacement
  COMPRESSED pt_2_seq_id {
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '11000'                               [ 5 ];
```

```
    msn           =:= msn_lsb(7)                              [ 7 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 5) [ 5 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)         [ 7 ];
    timestamp     =:= inferred_scaled_field                   [ 0 ];
  }

  // UOR-2-ID-ext1 replacement (both TS and IP-ID)
  COMPRESSED pt_2_seq_both {
    ENFORCE(ts_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '11001'                                 [ 5 ];
    msn           =:= msn_lsb(7)                              [ 7 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 5) [ 5 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)         [ 7 ];
    ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 7)    [ 7 ];
    marker        =:= irregular(1)                            [ 1 ];
  }

  // UOR-2-TS replacement
  COMPRESSED pt_2_seq_ts {
    ENFORCE(ts_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '1101'                                  [ 4 ];
    msn           =:= msn_lsb(7)                              [ 7 ];
    ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 5)    [ 5 ];
    marker        =:= irregular(1)                            [ 1 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)         [ 7 ];
    ip_id         =:= inferred_sequential_ip_id               [ 0 ];
  }
}

//////////////////////////////////////////
// UDP-lite profile
//////////////////////////////////////////

udplite_baseheader(profile_value, outer_ip_flag, ip_id_behavior_value,
                   reorder_ratio_value, coverage_behavior_value)
{
  UNCOMPRESSED v4 {
    outer_headers  =:= baseheader_outer_headers        [ VARIABLE ];
    ip_version     =:= uncompressed_value(4, 4)        [  4 ];
    header_length  =:= uncompressed_value(4, 5)        [  4 ];
```

```
    tos_tc                                                  [  8 ];
    length         =:= inferred_ip_v4_length                [ 16 ];
    ip_id                                                   [ 16 ];
    rf             =:= uncompressed_value(1, 0)             [  1 ];
    df                                                      [  1 ];
    mf             =:= uncompressed_value(1, 0)             [  1 ];
    frag_offset    =:= uncompressed_value(13, 0)            [ 13 ];
    ttl_hopl                                                [  8 ];
    next_header                                             [  8 ];
    ip_checksum =:= inferred_ip_v4_header_checksum          [ 16 ];
    src_addr                                                [ 32 ];
    dest_addr                                               [ 32 ];
    extension_headers =:= baseheader_extension_headers      [ VARIABLE ];
    src_port                                                [ 16 ];
    dst_port                                                [ 16 ];
    checksum_coverage                                       [ 16 ];
    udp_checksum                                            [ 16 ];
  }

  UNCOMPRESSED v6 {
    ENFORCE(ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_RANDOM);
    outer_headers  =:= baseheader_outer_headers             [ VARIABLE ];
    ip_version     =:= uncompressed_value(4, 6)             [   4 ];
    tos_tc                                                  [   8 ];
    flow_label                                              [  20 ];
    payload_length =:= inferred_ip_v6_length                [  16 ];
    next_header                                             [   8 ];
    ttl_hopl                                                [   8 ];
    src_addr                                                [ 128 ];
    dest_addr                                               [ 128 ];
    extension_headers =:= baseheader_extension_headers      [ VARIABLE ];
    src_port                                                [  16 ];
    dst_port                                                [  16 ];
    checksum_coverage                                       [  16 ];
    udp_checksum                                            [  16 ];
    df    =:= uncompressed_value(0,0)                       [   0 ];
    ip_id =:= uncompressed_value(0,0)                       [   0 ];
  }

  CONTROL {
    ENFORCE(profile_value == PROFILE_UDPLITE_0108);
    ENFORCE(profile == profile_value);
    ENFORCE(coverage_behavior.UVALUE == coverage_behavior_value);
    ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
    ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
  }

  DEFAULT {
```

```
    ENFORCE(outer_ip_flag == 0);
    tos_tc            =:= static;
    dest_addr         =:= static;
    ttl_hopl          =:= static;
    src_addr          =:= static;
    df                =:= static;
    flow_label        =:= static;
    next_header       =:= static;
    src_port          =:= static;
    dst_port          =:= static;
    reorder_ratio     =:= static;
    ip_id_behavior_innermost =:= static;
  }

  // Replacement for UOR-2-ext3
  COMPRESSED co_common {
    ENFORCE(outer_ip_flag == outer_ip_indicator.CVALUE);
    discriminator        =:= '11111010'                    [ 8 ];
    ip_id_indicator      =:= irregular(1)                  [ 1 ];
    header_crc   =:= crc7(THIS.UVALUE, THIS.ULENGTH)       [ 7 ];
    flags_indicator      =:= irregular(1)                  [ 1 ];
    ttl_hopl_indicator   =:= irregular(1)                  [ 1 ];
    tos_tc_indicator     =:= irregular(1)                  [ 1 ];
    reorder_ratio        =:= irregular(2)                  [ 2 ];
    control_crc3         =:= control_crc3_encoding         [ 3 ];
    outer_ip_indicator : df : ip_id_behavior_innermost :
      coverage_behavior  =:=
      profile_8_flags_enc(flags_indicator.CVALUE,
      ip_version.UVALUE)                                   [ 0, 8 ];
    tos_tc =:= static_or_irreg(tos_tc_indicator.CVALUE, 8) [ 0, 8 ];
    ttl_hopl =:= static_or_irreg(ttl_hopl_indicator.CVALUE,
      ttl_hopl.ULENGTH)                                    [ 0, 8 ];
    msn                  =:= msn_lsb(8)                    [ 8 ];
    ip_id =:= ip_id_sequential_variable(ip_id_behavior_innermost.UVALUE,
      ip_id_indicator.CVALUE)                             [ 0, 8, 16 ];
  }

  // UO-0
  COMPRESSED pt_0_crc3 {
    discriminator =:= '0'                               [ 1 ];
    msn           =:= msn_lsb(4)                        [ 4 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)   [ 3 ];
    ip_id         =:= inferred_sequential_ip_id         [ 0 ];
  }

  // New format, Type 0 with strong CRC and more SN bits
  COMPRESSED pt_0_crc7 {
    discriminator =:= '100'                             [ 3 ];
```

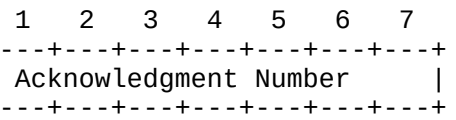
```
    msn           =:= msn_lsb(6)                        [ 6 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH) [ 7 ];
    ip_id         =:= inferred_sequential_ip_id       [ 0 ];
  }

  // UO-1-ID replacement (PT-1 only used for sequential)
  COMPRESSED pt_1_seq_id {
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '101'                                 [ 3 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)       [ 3 ];
    msn           =:= msn_lsb(6)                            [ 6 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 4) [ 4 ];
  }

  // UOR-2-ID replacement
  COMPRESSED pt_2_seq_id {
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '110'                                 [ 3 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 6) [ 6 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)       [ 7 ];
    msn           =:= msn_lsb(8)                            [ 8 ];
  }
}
```

6.9. Feedback Formats and Options

6.9.1. Feedback Formats

This section describes the feedback format for ROHCv2 profiles, using the formats described in Section 5.2.3 of [RFC4995].

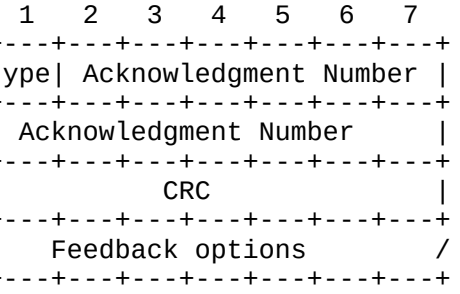
The Acknowledgment Number field of the feedback formats contains the least significant bits of the MSN (see Section 6.3.1) that corresponds to the reference header that is being acknowledged. A reference header is a header that has been successfully CRC-8 validated or CRC verified. If there is no reference header available, the feedback MUST carry an ACKNUMBER-NOT-VALID option. FEEDBACK-1



Acknowledgment Number: The eight least significant bits of the MSN.

A FEEDBACK-1 is an ACK. In order to send a NACK or a STATIC-NACK, FEEDBACK-2 must be used.

FEEDBACK-2



Acktype:

0 = ACK

1 = NACK

2 = STATIC-NACK

3 is reserved (MUST NOT be used for parsability)

Acknowledgment Number: The least significant bits of the MSN.

CRC: 8-bit CRC computed over the entire feedback payload including any CID fields but excluding the feedback type, the 'Size' field, and the 'Code' octet, using the polynomial defined in Section 5.3.1.1 of [RFC4995]. If the CID is given with an Add-CID octet, the Add-CID octet immediately precedes the FEEDBACK-1 or FEEDBACK-2 format. For purposes of computing the CRC, the CRC field is zero.

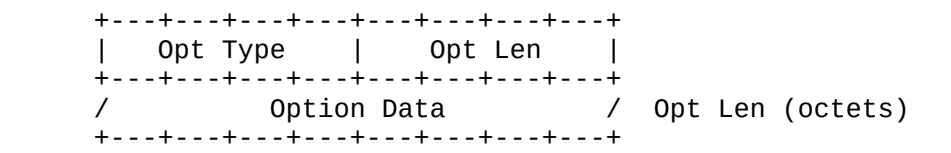
Feedback options: A variable number of feedback options, see Section 6.9.2. Options may appear in any order.

A FEEDBACK-2 of type NACK or STATIC-NACK is always implicitly an acknowledgment for a successfully decompressed packet, which corresponds to a packet whose LSBs match the Acknowledgment Number of the feedback element, unless the ACKNUMBER-NOT-VALID option (see Section 6.9.2.2) appears in the feedback element.

The FEEDBACK-2 format always carries a CRC and is thus more robust than the FEEDBACK-1 format. When receiving FEEDBACK-2, the compressor MUST verify the information by computing the CRC and comparing the result with the CRC carried in the feedback format. If the two are not identical, the feedback element MUST be discarded.

6.9.2. Feedback Options

A feedback option has variable length and the following general format:



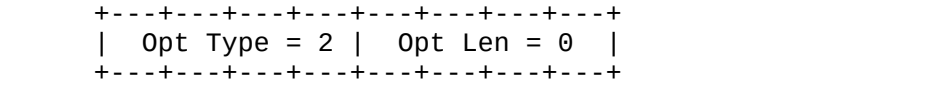
Opt Type: Unsigned integer that represents the type of the feedback option. Section 6.9.2.1 through Section 6.9.2.4 describes the ROHCv2 feedback options.

Opt Len: Unsigned integer that represents the length of the Option Data field, in octets.

Option Data: Feedback type specific data. Present if the value of the Opt Len field is set to a non-zero value.

6.9.2.1. The REJECT Option

The REJECT option informs the compressor that the decompressor does not have sufficient resources to handle the flow.

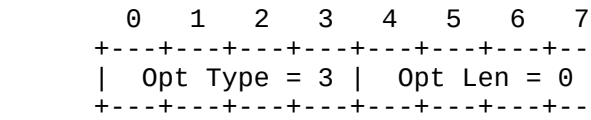


When receiving a REJECT option, the compressor MUST stop compressing the packet flow, and SHOULD refrain from attempting to increase the number of compressed packet flows for some time. The REJECT option

MUST NOT appear more than once in the FEEDBACK-2 format; otherwise, the compressor MUST discard the entire feedback element.

6.9.2.2. The ACKNUMBER-NOT-VALID Option

The ACKNUMBER-NOT-VALID option indicates that the Acknowledgment Number field of the feedback is not valid.

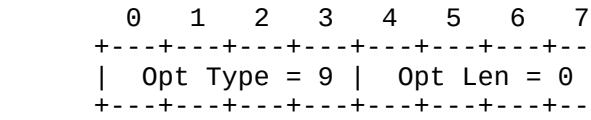


A compressor MUST NOT use the Acknowledgment Number of the feedback to find the corresponding sent header when this option is present. When this option is used, the Acknowledgment Number field of the FEEDBACK-2 format is set to zero. Consequently, a NACK or a STATIC-NACK feedback type sent with the ACKNUMBER-NOT-VALID option is equivalent to a STATIC-NACK with respect to the type of context repair requested by the decompressor.

The ACKNUMBER-NOT-VALID option MUST NOT appear more than once in the FEEDBACK-2 format; otherwise, the compressor MUST discard the entire feedback element.

6.9.2.3. The CONTEXT_MEMORY Option

The CONTEXT_MEMORY option informs the compressor that the decompressor does not have sufficient memory resources to handle the context of the packet flow, as the flow is currently compressed.

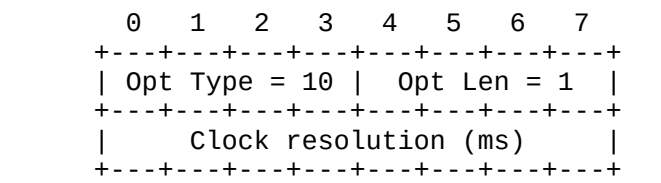


When receiving a CONTEXT_MEMORY option, the compressor SHOULD take actions to compress the packet flow in a way that requires less decompressor memory resources or stop compressing the packet flow. The CONTEXT_MEMORY option MUST NOT appear more than once in the FEEDBACK-2 format; otherwise, the compressor MUST discard the entire feedback element.

6.9.2.4. The CLOCK_RESOLUTION Option

The CLOCK_RESOLUTION option informs the compressor of the clock resolution of the decompressor. It also informs whether or not the decompressor supports timer-based compression of the RTP TS timestamp

(see Section 6.6.9). The CLOCK_RESOLUTION option is applicable per channel, i.e., it applies to any context associated with a profile for which the option is relevant between a compressor and decompressor pair.



Clock resolution: Unsigned integer that represents the clock resolution of the decompressor expressed in milliseconds.

The smallest clock resolution that can be indicated is 1 millisecond. The value zero has a special meaning: it indicates that the decompressor cannot do timer-based compression of the RTP Timestamp. The CLOCK_RESOLUTION option MUST NOT appear more than once in the FEEDBACK-2 format; otherwise, the compressor MUST discard the entire feedback element.

6.9.2.5. Unknown Option Types

If an option type other than those defined in this document is encountered, the compressor MUST discard the entire feedback element.

7. Security Considerations

Impairments such as bit errors on the received compressed headers, missing packets, and reordering between packets could cause the header decompressor to reconstitute erroneous packets, i.e., packets that do not match the original packet, but still have a valid IP, UDP (or UDP-Lite), and RTP headers, and possibly also valid UDP (or UDP-Lite) checksums.

The header compression profiles defined herein use an internal checksum for verification of reconstructed headers. This reduces the probability that a header decompressor delivers erroneous packets to upper layers without the error being noticed. In particular, the probability that consecutive erroneous packets are not detected by the internal checksum is close to zero.

This small but non-zero probability remains unchanged when integrity protection is applied after compression and verified before decompression, in the case where an attacker could discard or reorder packets between the compression endpoints.

The impairments mentioned above could be caused by a malfunctioning or malicious header compressor. Such corruption may be detected with end-to-end integrity mechanisms that will not be affected by the compression. Moreover, the internal checksum can also be useful in the case of malfunctioning compressors.

Denial-of-service attacks are possible if an intruder can introduce (for example) bogus IR or FEEDBACK packets onto the link and thereby cause compression efficiency to be reduced. However, an intruder having the ability to inject arbitrary packets at the link layer in this manner raises additional security issues that dwarf those related to the use of header compression.

8. IANA Considerations

The following ROHC profile identifiers have been assigned by the IANA for the profiles defined in this document:

Identifier	Profile
0x0101	ROHCv2 RTP
0x0102	ROHCv2 UDP
0x0103	ROHCv2 ESP
0x0104	ROHCv2 IP
0x0107	ROHCv2 RTP/UDP-Lite
0x0108	ROHCv2 UDP-Lite

9. Acknowledgements

The authors would like to thank Mark West, Robert Finking, Haipeng Jin, and Rohit Kapoor for serving as committed document reviewers, and also for constructive discussions during the development of this document. Thanks to Carl Knutsson for his extensive contribution to this specification, as well as to Jani Juvan and Anders Edqvist for useful comments and feedback. Thanks also to Elwyn Davies for his review as the General Area Review Team (Gen-ART) reviewer, and to Stephen Kent for his review on behalf of the IETF security directorate, during IETF last-call. Finally, thanks to the many people who have contributed to previous ROHC specifications and supported this effort.

10. References

10.1. Normative References

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.

[RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.

[RFC2004] Perkins, C., "Minimal Encapsulation within IP", RFC 2004, October 1996.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.

[RFC2784] Farinacci, D., Li, T., Hanks, S., Meyer, D., and P. Traina, "Generic Routing Encapsulation (GRE)", RFC 2784, March 2000.

[RFC2890] Dommety, G., "Key and Sequence Number Extensions to GRE", RFC 2890, September 2000.

[RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", STD 64, RFC 3550, July 2003.

[RFC3828] Larzon, L-A., Degermark, M., Pink, S., Jonsson, L-E., and G. Fairhurst, "The Lightweight User Datagram Protocol (UDP-Lite)", RFC 3828, July 2004.

[RFC4019] Pelletier, G., "RObust Header Compression (ROHC): Profiles for User Datagram Protocol (UDP) Lite", RFC 4019, April 2005.

[RFC4302] Kent, S., "IP Authentication Header", RFC 4302, December 2005.

[RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, December 2005.

[RFC4995] Jonsson, L-E., Pelletier, G., and K. Sandlund, "The RObust Header Compression (ROHC) Framework", RFC 4995, July 2007.

[RFC4997] Finking, R. and G. Pelletier, "Formal Notation for RObust Header Compression (ROHC-FN)", RFC 4997, July 2007.

10.2. Informative References

[RFC2675] Borman, D., Deering, S., and R. Hinden, "IPv6 Jumbograms", RFC 2675, August 1999.

[RFC3095] Bormann, C., Burmeister, C., Degermark, M., Fukushima, H., Hannu, H., Jonsson, L-E., Hakenberg, R., Koren, T., Le, K., Liu, Z., Martensson, A., Miyazaki, A., Svanbro, K., Wiebke, T., Yoshimura, T., and H. Zheng, "RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed", RFC 3095, July 2001.

[RFC3843] Jonsson, L-E. and G. Pelletier, "RObust Header Compression (ROHC): A Compression Profile for IP", RFC 3843, June 2004.

[RFC4224] Pelletier, G., Jonsson, L-E., and K. Sandlund, "RObust Header Compression (ROHC): ROHC over Channels That Can Reorder Packets", RFC 4224, January 2006.

Appendix A. Detailed Classification of Header Fields

Header compression is possible due to the fact that most header fields do not vary randomly from packet to packet. Many of the fields exhibit static behavior or change in a more or less predictable way. When designing a header compression scheme, it is of fundamental importance to understand the behavior of the fields in detail.

In this appendix, all fields in the headers compressible by these profiles are classified and analyzed. The analysis is based on behavior for the types of traffic that are expected to be the most frequently compressed (e.g., RTP field behavior is based on voice and/or video traffic behavior).

Fields are classified as belonging to one of the following classes:

INFERRED - These fields contain values that can be inferred from other values, for example the size of the frame carrying the packet, and thus do not have to be included in compressed packets.

STATIC - These fields are expected to be constant throughout the lifetime of the flow; in general, it is sufficient to design a compressed format so that these fields are only updated by IR packets.

STATIC-DEF - These fields are expected to be constant throughout the lifetime of the flow and their values can be used to define a flow. They are only sent in IR packets.

STATIC-KNOWN - These fields are expected to have well-known values and therefore do not need to be communicated at all.

SEMISTATIC - These fields are unchanged most of the time. However, occasionally the value changes but will revert to its original value. For ROHCv2, the values of such fields do not need to be possible to change with the smallest compressed packet formats, but should be possible to change via slightly larger compressed packets.

RARELY CHANGING (RACH) - These are fields that change their values occasionally and then keep their new values. For ROHCv2, the values of such fields do not need to be possible to change with the smallest compressed packet formats, but should be possible to change via slightly larger compressed packets.

IRREGULAR - These are the fields for which no useful change pattern can be identified and should be transmitted uncompressed in all compressed packets.

PATTERN - These are fields that change between each packet, but change in a predictable pattern.

A.1. IPv4 Header Fields

Field	Class
Version	STATIC-KNOWN
Header Length	STATIC-KNOWN
Type Of Service	RACH
Packet Length	INFERRED
Identification	
Sequential	PATTERN
Seq. swap	PATTERN
Random	IRREGULAR
Zero	STATIC
Reserved flag	STATIC-KNOWN
Don't Fragment flag	RACH
More Fragments flag	STATIC-KNOWN
Fragment Offset	STATIC-KNOWN
Time To Live	RACH
Protocol	STATIC-DEF
Header Checksum	INFERRED
Source Address	STATIC-DEF
Destination Address	STATIC-DEF

Version

The version field states which IP version is used and is set to the value four.

Header Length

As long as no options are present in the IP header, the header length is constant with the value five. If there are options, the field could be RACH or STATIC-DEF, but only option-less headers are compressed by ROHCv2 profiles. The field is therefore classified as STATIC-KNOWN.

Type Of Service

For the type of flows compressed by the ROHCv2 profiles, the DSCP (Differentiated Services Code Point) and ECN (Explicit Congestion Notification) fields are expected to change relatively seldom.

Packet Length

Information about packet length is expected to be provided by the link layer. The field is therefore classified as INFERRED.

IPv4 Identification

The Identification field (IP-ID) is used to identify what fragments constitute a datagram when reassembling fragmented datagrams. The IPv4 specification does not specify exactly how this field is to be assigned values, only that each packet should get an IP-ID that is unique for the source-destination pair and protocol for the time the datagram (or any of its fragments) could be alive in the network. This means that assignment of IP-ID values can be done in various ways, but the expected behaviors have been separated into four classes.

Sequential

In this behavior, the IP-ID is expected to increment by one for most packets, but may increment by a value larger than one, depending on the behavior of the transmitting IPv4 stack.

Sequential Swapped

When using this behavior, the IP-ID behaves as in the Sequential behavior, but the two bytes of IP-ID are byte-swapped. Therefore, the IP-ID can be swapped before compression to make it behave exactly as the Sequential behavior.

Random

Some IP stacks assign IP-ID values using a pseudo-random number generator. There is thus no correlation between the ID values of subsequent datagrams, and therefore there is no way to predict the IP-ID value for the next datagram. For header compression purposes, this means that the IP-ID field needs to be sent uncompressed with each datagram, resulting in two extra octets of header.

Zero

This behavior, although not a legal implementation of IPv4, is sometimes seen in existing IPv4 stacks. When this behavior is used, all IP packets have the IP-ID value set to zero.

Flags

The Reserved flag must be set to zero and is therefore classified as STATIC-KNOWN. The Don't Fragment (DF) flag changes rarely and is therefore classified as RACH. Finally, the More Fragments (MF) flag is expected to be zero because IP fragments will not be compressed by ROHC and is therefore classified as STATIC-KNOWN.

Fragment Offset

Under the assumption that no fragmentation occurs, the fragment offset is always zero and is therefore classified as STATIC-KNOWN.

Time To Live

The Time To Live field is expected to be constant during the lifetime of a flow or to alternate between a limited number of values due to route changes.

Protocol

This field will have the same value in all packets of a flow and is therefore classified as STATIC-DEF.

Header Checksum

The header checksum protects individual hops from processing a corrupted header. When almost all IP header information is compressed away, there is no point in having this additional checksum; instead, it can be regenerated at the decompressor side. The field is therefore classified as INFERRED.

Source and Destination addresses

These fields are part of the definition of a flow and must thus be constant for all packets in the flow.

A.2. IPv6 Header Fields

Field	Class
Version	STATIC-KNOWN
Traffic Class	RACH
Flow Label	STATIC-DEF
Payload Length	INFERRED
Next Header	STATIC-DEF
Hop Limit	RACH
Source Address	STATIC-DEF
Destination Address	STATIC-DEF

Version

The version field states which IP version is used and is set to the value six.

Traffic Class

For the type of flows compressed by the ROHCv2 profiles, the DSCP and ECN fields are expected to change relatively seldom.

Flow Label

This field may be used to identify packets belonging to a specific flow. If it is not used, the value should be set to zero. Otherwise, all packets belonging to the same flow must have the same value in this field. The field is therefore classified as STATIC-DEF.

Payload Length

Information about packet length (and, consequently, payload length) is expected to be provided by the link layer. The field is therefore classified as INFERRED.

Next Header

This field will have the same value in all packets of a flow and is therefore classified as STATIC-DEF.

Hop Limit

The Hop Limit field is expected to be constant during the lifetime of a flow or to alternate between a limited number of values due to route changes.

Source and Destination addresses

These fields are part of the definition of a flow and must thus be constant for all packets in the flow. The fields are therefore classified as STATIC-DEF.

A.3. UDP Header Fields

Field	Class
Source Port	STATIC-DEF
Destination Port	STATIC-DEF
Length	INFERRED
Checksum	
Disabled	STATIC
Enabled	IRREGULAR

Source and Destination ports

These fields are part of the definition of a flow and must thus be constant for all packets in the flow.

Length

Information about packet length is expected to be provided by the link layer. The field is therefore classified as INFERRED.

Checksum

The checksum can be optional. If disabled, its value is constantly zero and can be compressed away. If enabled, its value depends on the payload, which for compression purposes is equivalent to it changing randomly with every packet.

A.4. UDP-Lite Header Fields

Field	Class
Source Port	STATIC-DEF
Destination Port	STATIC-DEF
Checksum Coverage	
Zero	STATIC-DEF
Constant	INFERRED
Variable	IRREGULAR
Checksum	IRREGULAR

Source and Destination Port

These fields are part of the definition of a flow and must thus be constant for all packets in the flow.

Checksum Coverage

The Checksum Coverage field may behave in different ways: it may have a value of zero, it may be equal to the datagram length, or it may have any value between eight octets and the length of the datagram. From a compression perspective, this field is expected to either be entirely predictable (for the cases where it follows the same behavior as the UDP Length field or where it takes on a constant value) or to change randomly for each packet (making the value unpredictable from a header-compression perspective). For all cases, the behavior itself is not expected to change for this field during the lifetime of a packet flow, or to change relatively seldom.

Checksum

The information used for the calculation of the UDP-Lite checksum is governed by the value of the checksum coverage and minimally includes the UDP-Lite header. The checksum is a changing field that must always be sent as-is.

A.5. RTP Header Fields

Field	Class
Version	STATIC-KNOWN
Padding	RACH
Extension	RACH
CSRC Counter	RACH
Marker	SEMISTATIC
Payload Type	RACH
Sequence Number	PATTERN
Timestamp	PATTERN
SSRC	STATIC-DEF
CSRC	RACH

Version

This field is expected to have the value two and the field is therefore classified as STATIC-KNOWN.

Padding

The use of this field is application-dependent, but when payload padding is used, it is likely to be present in most or all packets. The field is classified as RACH to allow for the case where the value of this field changes.

Extension

If RTP extensions are used by the application, these extensions are often present in all packets, although the use of extensions is infrequent. To allow efficient compression of a flow using extensions in only a few packets, this field is classified as RACH.

CSRC Count

This field indicates the number of CSRC items present in the CSRC list. This number is expected to be mostly constant on a packet-to-packet basis and when it changes, change by small amounts. As long as no RTP mixer is used, the value of this field will be zero.

Marker

For audio, the marker bit should be set only in the first packet of a talkspurt, while for video, it should be set in the last packet of every picture. This means that in both cases the RTP marker is classified as SEMISTATIC.

Payload Type

Applications could adapt to congestion by changing payload type and/or frame sizes, but that is not expected to happen frequently, so this field is classified as RACH.

RTP Sequence Number

The RTP Sequence Number will be incremented by one for each packet sent.

Timestamp

In the audio case:

As long as there are no pauses in the audio stream, the RTP Timestamp will be incremented by a constant value, which corresponds to the number of samples in the speech frame. It will thus mostly follow the RTP Sequence Number. When there has been a silent period and a new talkspurt begins, the timestamp will jump in proportion to the length of the silent period. However, the increment will probably be within a relatively limited range.

In the video case:

Between two consecutive packets, the timestamp will either be unchanged or increase by a multiple of a fixed value corresponding to the picture clock frequency. The timestamp can also decrease by a multiple of the fixed value for certain coding schemes. The change in timestamp value, expressed as a multiple of the picture clock frequency, is in most cases within a limited range.

SSRC

This field is part of the definition of a flow and must thus be constant for all packets in the flow. The field is therefore classified as STATIC-DEF.

Contributing Sources (CSRC)

The participants in a session, who are identified by the CSRC fields, are usually expected to be unchanged on a packet-to-packet basis, but will infrequently change by a few additions and/or removals.

A.6. ESP Header Fields

Field	Class
SPI	STATIC-DEF
Sequence Number	PATTERN

SPI

This field is used to identify a distinct flow between two IPsec peers and it changes rarely; therefore, it is classified as STATIC-DEF.

ESP Sequence Number

The ESP Sequence Number will be incremented by one for each packet sent.

A.7. IPv6 Extension Header Fields

Field	Class
Next Header	STATIC-DEF
Ext Hdr Len	
Routing	STATIC-DEF
Hop-by-hop	STATIC
Destination	STATIC
Options	
Routing	STATIC-DEF
Hop-by-hop	RACH
Destination	RACH

Next Header

This field will have the same value in all packets of a flow and is therefore classified as STATIC-DEF.

Ext Hdr Len

For the Routing header, it is expected that the length will remain constant for the duration of the flow, and that a change in the length should be classified as a new flow by the ROHC compressor. For Hop-by-hop and Destination options headers, the length is expected to remain static, but can be updated by an IR packet.

Options

For the Routing header, it is expected that the option content will remain constant for the duration of the flow, and that a change in the routing information should be classified as a new flow by the ROHC compressor. For Hop-by-hop and Destination options headers, the options are expected to remain static, but can be updated by an IR packet.

A.8. GRE Header Fields

Field	Class
C flag	STATIC
K flag	STATIC
S flag	STATIC
R flag	STATIC-KNOWN
Reserved0, Version	STATIC-KNOWN
Protocol	STATIC-DEF
Checksum	IRREGULAR
Reserved	STATIC-KNOWN
Sequence Number	PATTERN
Key	STATIC-DEF

Flags

The four flag bits are not expected to change for the duration of the flow, and the R flag is expected to always be set to zero.

Reserved0, Version

Both of these fields are expected to be set to zero for the duration of any flow.

Protocol

This field will have the same value in all packets of a flow and is therefore classified as STATIC-DEF.

Checksum

When the checksum field is present, it is expected to behave unpredictably.

Reserved

When present, this field is expected to be set to zero.

Sequence Number

When present, the Sequence Number increases by one for each packet.

Key

When present, the Key field is used to define the flow and does not change.

A.9. MINE Header Fields

Field	Class
Protocol	STATIC-DEF
S bit	STATIC-DEF
Reserved	STATIC-KNOWN
Checksum	INFERRED
Source Address	STATIC-DEF
Destination Address	STATIC-DEF

Protocol

This field will have the same value in all packets of a flow and is therefore classified as STATIC-DEF.

S bit

The S bit is not expected to change during a flow.

Reserved

The reserved field is expected to be set to zero.

Checksum

The header checksum protects individual routing hops from processing a corrupted header. Since all fields of this header are compressed away, there is no need to include this checksum in compressed packets and it can be regenerated at the decompressor side.

Source and Destination Addresses

These fields can be used to define the flow and are not expected to change.

A.10. AH Header Fields

Field	Class
Next Header	STATIC-DEF
Payload Length	STATIC
Reserved	STATIC-KNOWN
SPI	STATIC-DEF
Sequence Number	PATTERN
ICV	IRREGULAR

Next Header

This field will have the same value in all packets of a flow and is therefore classified as STATIC-DEF.

Payload Length

It is expected that the length of the header is constant for the duration of the flow.

Reserved

The value of this field will be set to zero.

SPI

This field is used to identify a specific flow and only changes when the sequence number wraps around, and is therefore classified as STATIC-DEF.

Sequence Number

The Sequence Number will be incremented by one for each packet sent.

ICV

The ICV is expected to behave unpredictably and is therefore classified as IRREGULAR.

Appendix B. Compressor Implementation Guidelines

This section describes some guiding principles for implementing a ROHCv2 compressor with focus on how to efficiently select appropriate packet formats. The text in this appendix should be considered guidelines; it does not define any normative requirement on how ROHCv2 profiles are implemented.

B.1. Reference Management

The compressor usually maintains a sliding window of reference headers, which contains as many references as needed for the optimistic approach. Each reference contains a description of which changes occurred in the flow between two consecutive headers in the flow, and a new reference is inserted into the window each time a packet is compressed by this context. A reference may for example be implemented as a stored copy of the uncompressed header being represented. When the compressor is confident that a specific reference is no longer used by the decompressor (for example by using the optimistic approach or feedback received), the reference is removed from the sliding window.

B.2. Window-based LSB Encoding (W-LSB)

Section 5.1.1 describes how the optimistic approach impacts the packet format selection for the compressor. Exactly how the compressor selects a packet format is up to the implementation to decide, but the following is an example of how this process can be performed for lsb-encoded fields through the use of Window-based LSB encoding (W-LSB).

With W-LSB encoding, the compressor uses a number of references (a window) from its context. What references to use is determined by its optimistic approach. The compressor extracts the value of the field to be W-LSB encoded from each reference in the window, and finds the maximum and minimum values. Once it determines these values, the compressor uses the assumption that the decompressor has a value for this field within the range given by these boundaries

(inclusively) as its reference. The compressor can then select a number of LSBs from the value to be compressed, so that the LSBs can be decompressed regardless of whether the decompressor uses the minimum value, the maximum value or any other value in the range of possible references.

B.3. W-LSB Encoding and Timer-based Compression

Section 6.6.9 defines decompressor behavior for timer-based RTP timestamp compression. This section gives guidelines on how the compressor should determine the number of LSB bits it should send for the timestamp field. When using timer-based compression, this number depends on the sum of the jitter before the compressor and the jitter between the compressor and decompressor.

The jitter before the compressor can be estimated using the following computation:

```
Max_Jitter_BC =
     max {|(T_n - T_j) - ((a_n - a_j) / time_stride)|,
        for all headers j in the sliding window}
```

where (T_n - T_j) is the difference in the timestamp between the currently compressed header and a reference header and (a_n - a_j) is the difference in arrival time between those same two headers.

In addition to this, the compressor needs to estimate an upper bound for the jitter between the compressor and decompressor (Max_Jitter_CD). This information may for example come from lower layers.

A compressor implementation can determine whether the difference in clock resolution between the compressor and decompressor induces an error when performing integer arithmetics; it can then treat this error as additional jitter.

After obtaining estimates for the jitters, the number of bits needed to transmit is obtained using the following calculation:

```
ceiling(log2(2 * (Max_Jitter_BC + Max_Jitter_CD + 2) + 1))
```

This number is then used to select a packet format that contains at least this many scaled timestamp bits.

Authors' Addresses

Ghyslain Pelletier
Ericsson
Box 920
Lulea SE-971 28
Sweden

Phone: +46 (0) 8 404 29 43
EMail: ghyslain.pelletier@ericsson.com

Kristofer Sandlund
Ericsson
Box 920
Lulea SE-971 28
Sweden

Phone: +46 (0) 8 404 41 58
EMail: kristofer.sandlund@ericsson.com

Full Copyright Statement

Intellectual Property