Internet Engineering Task Force (IETF)
Request for Comments: 6846
Obsoletes: 4996
Category: Standards Track
ISSN: 2070-1721

G. Pelletier
InterDigital Communications
K. Sandlund
Ericsson
L-E. Jonsson

M. West
Siemens/Roke Manor

January 2013

RObust Header Compression (ROHC): A Profile for TCP/IP (ROHC-TCP)

Abstract

This document specifies a RObust Header Compression (ROHC) profile for compression of TCP/IP packets. The profile, called ROHC-TCP, provides efficient and robust compression of TCP headers, including frequently used TCP options such as selective acknowledgments (SACKs) and Timestamps.

ROHC-TCP works well when used over links with significant error rates and long round-trip times. For many bandwidth-limited links where header compression is essential, such characteristics are common.

This specification obsoletes RFC 4996. It fixes a technical issue with the SACK compression and clarifies other compression methods used.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6846.

Table of Contents

1. Introduction

There are several reasons to perform header compression on low- or medium-speed links for TCP/IP traffic, and these have already been discussed in [RFC2507]. Additional considerations that make robustness an important objective for a TCP [RFC0793] compression scheme are introduced in [RFC4163]. Finally, existing TCP/IP header compression schemes ([RFC1144], [RFC2507]) are limited in their handling of the TCP options field and cannot compress the headers of handshaking packets (SYNs and FINs).

It is thus desirable for a header compression scheme to be able to handle loss on the link between the compression and decompression points as well as loss before the compression point. The header compression scheme also needs to consider how to efficiently compress short-lived TCP transfers and TCP options, such as selective acknowledgments (SACK) ([RFC2018], [RFC2883]) and Timestamps ([RFC1323]). TCP options that may be less frequently used do not necessarily need to be compressed by the protocol, and instead can be passed transparently without reducing the overall compression efficiency of other parts of the TCP header.

The Robust Header Compression (ROHC) Working Group has developed a header compression framework on top of which various profiles can be defined for different protocol sets, or for different compression strategies. This document defines a TCP/IP compression profile for the ROHC framework [RFC5795], compliant with the requirements listed in [RFC4163].

Specifically, it describes a header compression scheme for TCP/IP header compression (ROHC-TCP) that is robust against packet loss and that offers enhanced capabilities, in particular for the compression of header fields including TCP options. The profile identifier for TCP/IP compression is 0x0006.

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 [RFC2119].

This document reuses some of the terminology found in [RFC5795]. In addition, this document uses or defines the following terms:

Base context

The base context is a context that has been validated by both the compressor and the decompressor. A base context can be used as the reference when building a new context using replication.

Base Context Identifier (Base CID)

The Base CID is the CID that identifies the base context, from which information needed for context replication can be extracted.

Base header

The Base header is a compressed representation of the innermost IP and TCP headers of the uncompressed packet.

Chaining of items

A chain groups fields based on similar characteristics. ROHC-TCP defines chain items for static, dynamic, replicable, or irregular fields. Chaining is done by appending an item for each header, e.g., to the chain in their 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 ROHC-TCP defines a compressed format.

Context Replication (CR)

Context replication is the mechanism that establishes and initializes a new context based on another existing valid context (a base context). This mechanism is introduced to reduce the overhead of the context establishment procedure, and is especially useful for compression of multiple short-lived TCP connections that may be occurring simultaneously or near-simultaneously.

ROHC-TCP packet types

ROHC-TCP uses three different packet types: the Initialization and Refresh (IR) packet type, the Context Replication (IR-CR) packet type, and the Compressed packet (CO) type.

Short-lived TCP transfer

Short-lived TCP transfers refer to TCP connections transmitting only small amounts of packets for each single connection.

3. Background

This section provides some background information on TCP/IP header compression. The fundamentals of general header compression can be found in [RFC5795]. In the following subsections, two existing TCP/IP header compression schemes are first described along with a discussion of their limitations, followed by the classification of TCP/IP header fields. Finally, some of the characteristics of short-lived TCP transfers are summarized.

A behavior analysis of TCP/IP header fields is found in [RFC4413].

3.1. Existing TCP/IP Header Compression Schemes

Compressed TCP (CTCP) and IP Header Compression (IPHC) are two different schemes that may be used to compress TCP/IP headers. Both schemes transmit only the differences from the previous header in order to reduce the size of the TCP/IP header.

The CTCP [RFC1144] compressor detects transport-level retransmissions and sends a header that updates the context completely when they occur. While CTCP works well over reliable links, it is vulnerable when used over less reliable links as even a single packet loss results in loss of synchronization between the compressor and the decompressor. This in turn leads to the TCP receiver discarding all remaining packets in the current window because of a checksum error. This effectively prevents the TCP fast retransmit algorithm [RFC5681] from being triggered. In such a case, the compressor must wait until TCP times out and retransmits a packet to resynchronize.

To reduce the errors due to the inconsistent contexts between compressor and decompressor when compressing TCP, IPHC [RFC2507] improves somewhat on CTCP by augmenting the repair mechanism of CTCP with a local repair mechanism called TWICE and with a link-layer mechanism based on negative acknowledgments to request a header that updates the context.

The TWICE algorithm assumes that only the Sequence Number field of TCP segments is changing with the deltas between consecutive packets being constant in most cases. This assumption is, however, not always true, especially when TCP Timestamps and SACK options are used.

The full header request mechanism requires a feedback channel that may be unavailable in some circumstances. This channel is used to explicitly request that the next packet be sent with an uncompressed header to allow resynchronization without waiting for a TCP timeout.

In addition, this mechanism does not perform well on links with long round-trip times.

Both CTCP and IPHC are also limited in their handling of the TCP options field. For IPHC, any change in the options field (caused by Timestamps or SACK, for example) renders the entire field uncompressible, while for CTCP, such a change in the options field effectively disables TCP/IP header compression altogether.

Finally, existing TCP/IP compression schemes do not compress the headers of handshaking packets (SYNs and FINs). Compressing these packets may greatly improve the overall header compression ratio for the cases where many short-lived TCP connections share the same channel.

3.2. Classification of TCP/IP Header Fields

Header compression is possible due to the fact that there is much redundancy between header field values within packets, especially between consecutive packets. To utilize these properties for TCP/IP header compression, it is important to understand the change patterns of the various header fields.

All fields of the TCP/IP packet header have been classified in detail in [RFC4413]. The main conclusion is that most of the header fields can easily be compressed away since they seldom or never change. The following fields do, however, require more sophisticated mechanisms:

- IPv4 Identification (16 bits) - IP-ID
- TCP Sequence Number (32 bits) - SN
- TCP Acknowledgment Number (32 bits)
- TCP Reserved (4 bits)
- TCP ECN flags (2 bits) - ECN
- TCP Window (16 bits)
- TCP Options
  - o Maximum Segment Size (32 bits) - MSS
  - o Window Scale (24 bits) - WSCALE
  - o SACK Permitted (16 bits)
  - o TCP SACK (80, 144, 208, or 272 bits) - SACK
  - o TCP Timestamp (80 bits) - TS

The assignment of IP-ID values can be done in various ways, usually one of sequential, sequential jump, or random, as described in Section 4.1.3 of [RFC4413]. Some IPv4 stacks do use a sequential assignment when generating IP-ID values but do not transmit the contents of this field in network byte order; instead, it is sent with the two octets reversed. In this case, the compressor can

compress the IP-ID field after swapping the bytes. Consequently, the decompressor also swaps the bytes of the IP-ID after decompression to regenerate the original IP-ID. With respect to TCP compression, the analysis in [RFC4413] reveals that there is no obvious candidate among the TCP fields suitable to infer the IP-ID.

The change pattern of several TCP fields (Sequence Number, Acknowledgment Number, Window, etc.) is very hard to predict. Of particular importance to a TCP/IP header compression scheme is the understanding of the sequence and acknowledgment numbers [RFC4413].

Specifically, the TCP Sequence Number can be anywhere within a range defined by the TCP Window at any point on the path (i.e., wherever a compressor might be deployed). Missing packets or retransmissions can cause the TCP Sequence Number to fluctuate within the limits of this window. The TCP Window also bounds the jumps in acknowledgment number.

Another important behavior of the TCP/IP header is the dependency between the sequence number and the acknowledgment number. TCP connections can be either near-symmetrical or show a strong asymmetrical bias with respect to the data traffic. In the latter case, the TCP connections mainly have one-way traffic (Web browsing and file downloading, for example). This means that on the forward path (from server to client), only the sequence number is changing while the acknowledgment number remains constant for most packets; on the backward path (from client to server), only the acknowledgment number is changing and the sequence number remains constant for most packets. A compression scheme for TCP should thus have packet formats suitable for either cases, i.e., packet formats that can carry either only sequence number bits, only acknowledgment number bits, or both.

In addition, TCP flows can be short-lived transfers. Short-lived TCP transfers will degrade the performance of header compression schemes that establish a new context by initially sending full headers. Multiple simultaneous or near simultaneous TCP connections may exhibit much similarity in header field values and context values among each other, which would make it possible to reuse information between flows when initializing a new context. A mechanism to this end, context replication [RFC4164], makes the context establishment step faster and more efficient, by replicating part of an existing context to a new flow. The conclusion from [RFC4413] is that part of the IP sub-context, some TCP fields, and some context values can be replicated since they seldom change or change with only a small jump.

ROHC-TCP also compresses the following headers: IPv6 Destination Options header [RFC2460], IPv6 Routing header [RFC2460], IPv6 Hop-by-Hop Options header [RFC2460], Authentication Header (AH) [RFC4302], Generic Routing Encapsulation (GRE) [RFC2784][RFC2890], and the Minimal Encapsulation (MINE) header [RFC2004].

Headers specific to Mobile IP (for IPv4 or IPv6) do not receive any special treatment in this document, for reasons similar to those described in [RFC3095].

4. Overview of the TCP/IP Profile (Informative)

4.1. General Concepts

ROHC-TCP uses the ROHC protocol as described in [RFC5795]. ROHC-TCP supports context replication as defined in [RFC4164]. Context replication can be particularly useful for short-lived TCP flows [RFC4413].

4.2. Compressor and Decompressor Interactions

4.2.1. Compressor Operation

Header compression with ROHC can be conceptually characterized as the interaction of a compressor with a decompressor state machine. The compressor’s task is to minimally send the information needed to successfully decompress a packet, based on a certain confidence regarding the state of the decompressor context.

For ROHC-TCP compression, 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. Alternatively, the compressor may also support Context Replication (CR) and use IR-CR packets [RFC4164], which attempts to reuse context information related to another flow.

The compressor can then adjust the compression level based on its confidence that the decompressor has the necessary information to successfully process the Compressed (CO) packets that it selects. In other words, the task of the compressor is to ensure that the decompressor operates in the state that allows decompression of the most efficient CO packet(s), and to allow the decompressor to move to that state as soon as possible otherwise.

4.2.2. Decompressor Feedback

The ROHC-TCP profile can be used in environments with or without feedback capabilities from decompressor to compressor. ROHC-TCP, 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 ROHC-TCP context, this channel will be used during the entire compression operation for that context. If the feedback channel disappears, compression should be restarted.

The reception of either positive acknowledgments (ACKs) or negative acknowledgments (NACKs) establishes the feedback channel from the decompressor for the context for which the feedback was received. Once there is an established feedback channel for a specific context, the compressor should make use of this feedback to estimate the current state of the decompressor. This helps in increasing the compression efficiency by providing the information needed for the compressor to achieve the necessary confidence level.

The ROHC-TCP feedback mechanism is limited in its applicability by the number of (least significant bit (LSB) encoded) master sequence number (MSN) (see Section 6.1.1) bits used in the FEEDBACK-2 format (see Section 8.3). It is not suitable for a decompressor to use feedback altogether where the MSN bits in the feedback could wrap around within one round-trip time. Instead, unidirectional operation -- where the compressor periodically sends larger context-updating packets -- is more appropriate.

4.3. Packet Formats and Encoding Methods

The packet formats and encoding methods used for ROHC-TCP are defined using the formal notation [RFC4997]. The formal notation is used to provide an unambiguous representation of the packet formats and a clear definition of the encoding methods.

4.3.1. Compressing TCP Options

The TCP options in ROHC-TCP are compressed using a list compression encoding that allows option content to be established so that TCP options can be added to the context without having to send all TCP options uncompressed.

4.3.2. Compressing Extension Headers

ROHC-TCP compresses the extension headers as listed in Section 3.2. These headers are treated exactly as other headers and thus have a static chain, a dynamic chain, an irregular chain, and a chain for context replication (Section 6.2).

This means that headers appearing in or disappearing from the flow being compressed will lead to changes to the static chain. However, the change pattern of extension headers is not deemed to impair compression efficiency with respect to this design strategy.

4.4. Expected Compression Ratios with ROHC-TCP

The following table illustrates typical compression ratios that can be expected when using ROHC-TCP and IPHC [RFC2507].

The figures in the table assume that the compression context has already been properly initialized. For the TS option, the Timestamp is assumed to change with small values. All TCP options include a suitable number of No Operation (NOP) options [RFC0793] for padding and/or alignment. Finally, in the examples for IPv4, a sequential IP-ID behavior is assumed.

	Total Header Size (octets)					
		ROHC-TCP		IPHC		
	Unc.	DATA	ACK	DATA	ACK	
IPv4+TCP+TS	52	8	8	18	18	
IPv4+TCP+TS	52	7	6	16	16	(1)
IPv6+TCP+TS	72	8	7	18	18	
IPv6+TCP+no opt	60	6	5	6	6	
IPv6+TCP+SACK	80	-	15	-	80	(2)
IPv6+TCP+SACK	80	-	9	-	26	(3)

(1) The payload size of the data stream is constant.
(2) The SACK option appears in the header, but was not present in the previous packet. Two SACK blocks are assumed.
(3) The SACK option appears in the header, and was also present in the previous packet (with different SACK blocks). Two SACK blocks are assumed.

The table below illustrates the typical initial compression ratios for ROHC-TCP and IPHC. The data stream in the example is assumed to be IPv4+TCP, with a sequential behavior for the IP-ID. The following options are assumed present in the SYN packet: TS, MSS, and WSCALE, with an appropriate number of NOP options.

	Total Header Size (octets)		
	Unc.	ROHC-TCP	IPHC
1st packet (SYN)	60	49	60
2nd packet	52	12	52

The figures in the table assume that the compressor has received an acknowledgment from the decompressor before compressing the second packet, which can be expected when feedback is used in ROHC-TCP.

This is because in the most common case, the TCP ACKs are expected to take the same return path, and because TCP does not send more packets until the TCP SYN packet has been acknowledged.

5. Compressor and Decompressor Logic (Normative)

5.1. Context Initialization

The static context of ROHC-TCP flows can be initialized in either of two ways:

1. By using an IR packet as in Section 7.1, where the profile number is 0x06 and the static chain ends with the static part of a TCP header.

2. By replicating an existing context using the mechanism defined by [RFC4164]. This is done with the IR-CR packet defined in Section 7.2, where the profile number is 0x06.

5.2. Compressor Operation

5.2.1. Compression Logic

The task of the compressor is to determine what data must be sent when compressing a TCP/IP packet, so that the decompressor can successfully reconstruct the original packet based on its current state. The selection of the type of compressed header to send thus depends on a number of factors, including:

- The change behavior of header fields in the flow, e.g., conveying the necessary information within the restrictions of the set of available packet formats.

- The compressor’s level of confidence regarding decompressor state, e.g., by selecting header formats updating the same type of information for a number of consecutive packets or from the reception of decompressor feedback (ACKs and/or NACKs).

- Additional robustness required for the flow, e.g., periodic refreshes of static and dynamic information using IR and IR-DYN packets when decompressor feedback is not expected.

The impact of these factors on the compressor’s packet type selection is described in more detail in the following subsections.

In this section, a "higher compression state" means that less data will be sent in compressed packets, i.e., smaller compressed headers are used, while a lower compression state means that a larger amount of data will be sent using larger compressed headers.

5.2.1.1. Optimistic Approach

The optimistic approach is the principle by which a compressor sends the same type of information for a number of packets (consecutively or not) until it is fairly confident that the decompressor has received the information. The optimistic approach is useful to ensure robustness when ROHC-TCP is used to compress packets over lossy links.

Therefore, if field X in the uncompressed packet changes value, the compressor MUST use a packet type that contains an encoding for field X until it has gained confidence that the decompressor has received at least one packet containing the new value for X. The compressor SHOULD choose a compressed format with the smallest header that can convey the changes needed to fulfill the optimistic approach condition used.

5.2.1.2. Periodic Context Refreshes

When the optimistic approach is used, there will always be a possibility of decompression failures since the decompressor may not have received sufficient information for correct decompression.

Therefore, until the decompressor has established a feedback channel, the compressor SHOULD periodically move to a lower compression state and send IR and/or IR-DYN packets. These refreshes can be based on timeouts, on the number of compressed packets sent for the flow, or any other strategy specific to the implementation. Once the feedback channel is established, the decompressor MAY stop performing periodic refreshes.

5.2.2. Feedback Logic

The semantics of feedback messages, acknowledgments (ACKs) and negative acknowledgments (NACKs or STATIC-NACKs), are defined in Section 5.2.4.1 of [RFC5795].

5.2.2.1. Optional Acknowledgments (ACKs)

The compressor MAY use acknowledgment feedback (ACKs) to move to a higher compression state.

Upon reception of an ACK for a context-updating packet, the compressor obtains confidence that the decompressor has received the acknowledged packet and that it has observed changes in the packet flow up to the acknowledged packet.

This functionality is optional, so a compressor MUST NOT expect to get such ACKs, even if a feedback channel is available and has been established for that flow.

5.2.2.2. Negative Acknowledgments (NACKs)

The compressor uses feedback from the decompressor to move to a lower compression state (NACKs).

On reception of a NACK feedback, the compressor SHOULD:

- assume that only the static part of the decompressor is valid, and
- re-send all dynamic information (via an IR or IR-DYN packet) the next time it compresses a packet for the indicated flow

unless it has confidence that information sent after the packet being acknowledged already provides a suitable response to the NACK feedback. In addition, the compressor MAY use a CO packet carrying a 7-bit Cyclic Redundancy Check (CRC) if it can determine with enough confidence what information provides a suitable response to the NACK feedback.

On reception of a STATIC-NACK feedback, the compressor SHOULD:

- assume that the decompressor has no valid context, and
- re-send all static and all dynamic information (via an IR packet) the next time it compresses a packet for the indicated flow

unless it has confidence that information sent after the packet that is being acknowledged already provides a suitable response to the STATIC-NACK feedback.

5.2.3. Context Replication

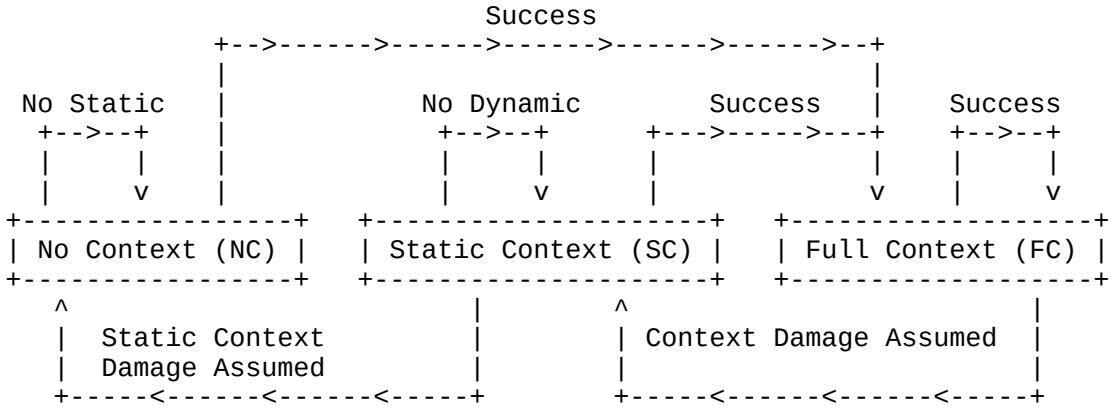
A compressor MAY support context replication by implementing the additional compression and feedback logic defined in [RFC4164].

5.3. Decompressor Operation

5.3.1. Decompressor States and Logic

The three states of the decompressor are No Context (NC), Static Context (SC), and Full Context (FC). The decompressor starts in its lowest compression state, the NC state. Successful decompression will always move the decompressor to the FC state. The decompressor state machine normally never leaves the FC state once it has entered this state; only repeated decompression failures will force the decompressor to transit downwards to a lower state.

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



5.3.1.1. Reconstruction and Verification

When decompressing an IR or an IR-DYN packet, the decompressor MUST validate the integrity of the received header using CRC-8 validation [RFC5795]. If validation fails, the packet MUST NOT be delivered to upper layers.

Upon receiving an IR-CR packet, the decompressor MUST perform the actions as specified in [RFC4164].

When decompressing other packet types (e.g., CO packets), the decompressor MUST validate the outcome of the decompression attempt using CRC verification [RFC5795]. If verification fails, a decompressor implementation MAY attempt corrective or repair measures on the packet, and the result of any attempt MUST be validated using the CRC verification; otherwise, the packet MUST NOT be delivered to upper layers.

When the CRC-8 validation or the CRC verification of the received header is successful, the decompressor SHOULD update its context with the information received in the current header; the decompressor then passes the reconstructed packet to the system's network layer. Otherwise, the decompressor context MUST NOT be updated.

If the received packet is older than the current reference packet, e.g., based on the master sequence number (MSN) in the compressed packet, the decompressor MAY refrain from updating the context using the information received in the current packet, even if the correctness of its header was successfully verified.

5.3.1.2. Detecting Context Damage

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 that are inferred from other fields).

The decompressor may assume that some or the entire context is invalid, following one or more failures to validate or verify a header 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 exact context entry 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 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.

Context Damage Detection

> The assumption of context damage means that the decompressor will not attempt decompression of a CO header that carries a 3-bit CRC, and only attempt decompression of IR, IR-DYN, or IR-CR 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 may attempt decompression. In other words, validity of the context refers to the ability of a decompressor to attempt or not attempt decompression of specific packet types.

5.3.1.3. No Context (NC) State

Initially, while working in the No Context (NC) state, the decompressor has not yet successfully decompressed a packet.

Allowing decompression:

In the NC state, only packets carrying sufficient information on the static fields (IR and IR-CR packets) can be decompressed; otherwise, the packet MUST NOT be decompressed and MUST NOT be delivered to upper layers.

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 decompression of an IR packet has failed, subject to the feedback rate limitation as described in Section 5.3.2.

Once a packet has been validated and decompressed correctly, the decompressor MUST transit to the FC state.

5.3.1.4. Static Context (SC) State

When the decompressor is in the Static Context (SC) state, only the static part of the decompressor context is valid.

From the SC state, the decompressor moves back to the NC state if static context damage is detected.

Allowing decompression:

In the SC state, packets carrying sufficient information on the dynamic fields covered by an 8-bit CRC (e.g., IR and IR-DYN) or CO packets covered by a 7-bit CRC can be decompressed; otherwise, the packet MUST NOT be decompressed and MUST NOT be delivered to upper layers.

Feedback logic:

In the SC state, the decompressor should send a STATIC-NACK if CRC validation of an IR/IR-DYN/IR-CR fails and static context damage is assumed. If any other packet type is received, the decompressor should send a NACK. Both of the above cases are subject to the feedback rate limitation as described in Section 5.3.2.

Once a packet has been validated and decompressed correctly, the decompressor MUST transit to the FC state.

5.3.1.5. Full Context (FC) State

In the Full Context (FC) state, both the static and the dynamic parts of the decompressor context are valid. From the FC state, the decompressor moves back to the SC state if context damage is detected.

Allowing decompression:

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

Feedback logic:

In the FC state, the decompressor should send a NACK if the decompression of any packet type fails and context damage is assumed, subject to the feedback rate limitation as described in Section 5.3.2.

5.3.2. Feedback Logic

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) establishes this channel.

Once the feedback channel is established, the decompressor is REQUIRED to continue sending NACKs or STATIC-NACKs for as long as the context is associated with the same profile, in this case with profile 0x0006, as per the logic defined for each state in Section 5.3.1.

The decompressor MAY send ACKs upon successful decompression of any packet type. In particular, when a packet carrying a significant context update is correctly decompressed, the decompressor MAY send an ACK.

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 to the same event.

5.3.3. Context Replication

ROHC-TCP supports context replication; therefore, the decompressor MUST implement the additional decompressor and feedback logic defined in [RFC4164].

6. Encodings in ROHC-TCP (Normative)

6.1. Control Fields in ROHC-TCP

In ROHC-TCP, a number of control fields are used by the decompressor in its interpretation of the format of the packets received from the compressor.

A control field is a field that is transmitted from the compressor to the decompressor, but is not part of the uncompressed header. Values for control fields can be set up in the context of both the compressor and the decompressor. Once established at the decompressor, the values of these fields should be kept until updated by another packet.

6.1.1. Master Sequence Number (MSN)

There is no field in the TCP header that can act as the master sequence number for TCP compression, as explained in [RFC4413], Section 5.6.

To overcome this problem, ROHC-TCP introduces a control field called the Master Sequence Number (MSN) field. The MSN field is created at the compressor, rather than using one of the fields already present in the uncompressed header. The compressor increments the value of the MSN by one for each packet that it sends.

The MSN field has the following two functions:

1. Differentiating between packets when sending feedback data.

2. Inferring the value of incrementing fields such as the IP-ID.

The MSN field is present in every packet sent by the compressor. The MSN is LSB encoded within the CO packets, and the 16-bit MSN is sent in full in IR/IR-DYN packets. The decompressor always sends the MSN as part of the feedback information. The compressor can later use the MSN to infer which packet the decompressor is acknowledging.

When the MSN is initialized, it SHOULD be initialized to a random value. The compressor should only initialize a new MSN for the initial IR or IR-CR packet sent for a CID that corresponds to a context that is not already associated with this profile. In other words, if the compressor reuses the same CID to compress many TCP flows one after the other, the MSN is not reinitialized but rather continues to increment monotonically.

For context replication, the compressor does not use the MSN of the base context when sending the IR-CR packet, unless the replication process overwrites the base context (i.e., Base CID == CID). Instead, the compressor uses the value of the MSN if it already exists in the ROHC-TCP context being associated with the new flow (CID); otherwise, the MSN is initialized to a new value.

6.1.2. IP-ID Behavior

The IP-ID field of the IPv4 header can have different change patterns. Conceptually, a compressor monitors changes in the value of the IP-ID field and selects encoding methods and packet formats that are the closest match to the observed change pattern.

ROHC-TCP defines different types of compression techniques for the IP-ID, to provide the flexibility to compress any of the behaviors it may observe for this field: sequential in network byte order (NBO), sequential byte-swapped, random (RND), or constant to a value of zero.

The compressor monitors changes in the value of the IP-ID field for a number of packets, to identify which one of the above listed compression alternatives is the closest match to the observed change pattern. The compressor can then select packet formats and encoding methods based on the identified field behavior.

If more than one level of IP headers is present, ROHC-TCP can assign a sequential behavior (NBO or byte-swapped) only to the IP-ID of the innermost IP header. This is because only this IP-ID can possibly have a sufficiently close correlation with the MSN (see also Section 6.1.1) to compress it as a sequentially changing field. Therefore, a compressor MUST NOT assign either the sequential (NBO) or the sequential byte-swapped behavior to tunneling headers.

The control field for the IP-ID behavior determines which set of packet formats will be used. These control fields are also used to determine the contents of the irregular chain item (see Section 6.2) for each IP header.

6.1.3. Explicit Congestion Notification (ECN)

When ECN [RFC3168] is used once on a flow, the ECN bits could change quite often. ROHC-TCP maintains a control field in the context to indicate whether or not ECN is used. This control field is transmitted in the dynamic chain of the TCP header, and its value can be updated using specific compressed headers carrying a 7-bit CRC.

When this control field indicates that ECN is being used, items of all IP and TCP headers in the irregular chain include bits used for ECN. To preserve octet-alignment, all of the TCP reserved bits are transmitted and, for outer IP headers, the entire Type of Service/ Traffic Class (TOS/TC) field is included in the irregular chain. When there is only one IP header present in the packet (i.e., no IP tunneling is used), this compression behavior allows the compressor to handle changes in the ECN bits by adding a single octet to the compressed header.

The reason for including the ECN bits of all IP headers in the compressed packet when the control field is set is that the profile needs to efficiently compress flows containing IP tunnels using the "full-functionality option" of Section 9.1 of [RFC3168]. For these flows, a change in the ECN bits of an inner IP header is propagated to the outer IP headers. When the "limited-functionality" option is used, the compressor will therefore sometimes send one octet more than necessary per tunnel header, but this has been considered a reasonable trade-off when designing this profile.

6.2. Compressed Header Chains

Some packet 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.

Chains are defined for all headers compressed by ROHC-TCP, as listed below. Also listed are the names of the encoding methods used to encode each of these protocol headers.

- TCP [RFC0793], encoding method: "tcp"
- IPv4 [RFC0791], encoding method: "ipv4"
- IPv6 [RFC2460], encoding method: "ipv6"
- AH [RFC4302], encoding method: "ah"
- GRE [RFC2784][RFC2890], encoding method: "gre"
- MINE [RFC2004], encoding method: "mine"
- IPv6 Destination Options header [RFC2460], encoding method: "ip_dest_opt"
- IPv6 Hop-by-Hop Options header [RFC2460], encoding method: "ip_hop_opt"
- IPv6 Routing header [RFC2460], encoding method: "ip_rout_opt"

Static chain:

The static chain consists of one item for each header of the chain of protocol headers to be compressed, starting from the outermost IP header and ending with a TCP header. In the formal description of the packet formats, this static chain item for each header is a format whose name is suffixed by "_static". The static chain is only used in IR packets.

Dynamic chain:

The dynamic chain consists of one item for each header of the chain of protocol headers to be compressed, starting from the outermost IP header and ending with a TCP header. The dynamic chain item for the TCP header also contains a compressed list of TCP options (see Section 6.3). In the formal description of the packet formats, the dynamic chain item for each header type is a format whose name is suffixed by "_dynamic". The dynamic chain is used in both IR and IR-DYN packets.

Replicate chain:

The replicate chain consists of one item for each header in the chain of protocol headers to be compressed, starting from the outermost IP header and ending with a TCP header. The replicate chain item for the TCP header also contains a compressed list of TCP options (see Section 6.3). In the formal description of the packet formats, the replicate chain item for each header type is a

format whose name is suffixed by "_replicate". Header fields that are not present in the replicate chain are replicated from the base context. The replicate chain is only used in the IR-CR packet.

Irregular chain:

The structure of the irregular chain is analogous to the structure of the static chain. For each compressed packet, the irregular chain is appended at the specified location in the general format of the compressed packets as defined in Section 7.3. This chain also includes the irregular chain items for TCP options as defined in Section 6.3.6, which are placed directly after the irregular chain item of the TCP header, and in the same order as the options appear in the uncompressed packet. In the formal description of the packet formats, the irregular chain item for each header type is a format whose name is suffixed by "_irregular". The irregular chain is used only in CO packets.

The format of the irregular chain for the innermost IP header differs from the format of outer IP headers, since this header is part of the compressed base header.

6.3. Compressing TCP Options with List Compression

This section describes in detail how list compression is applied to the TCP options. In the definition of the packet formats for ROHC-TCP, the most frequent TCP options have one encoding method each, as listed in the table below.

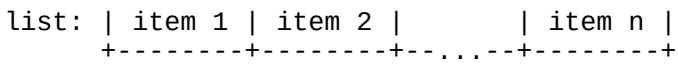
Option name	Encoding method name
NOP	tcp_opt_nop
EOL	tcp_opt_eol
MSS	tcp_opt_mss
WINDOW SCALE	tcp_opt_wscale
TIMESTAMP	tcp_opt_ts
SACK-PERMITTED	tcp_opt_sack_permitted
SACK	tcp_opt_sack
Generic options	tcp_opt_generic

Each of these encoding methods has an uncompressed format, a format suffixed by "_list_item" and a format suffixed by "_irregular". In some cases, a single encoding method may have multiple "_list_item"

or "_irregular" formats, in which case bindings inside these formats determine what format is used. This is further described in the following sections.

6.3.1. List Compression

The TCP options 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:



To compress this list, ROHC-TCP uses a list compression scheme, which compresses each of these items individually and combines them into a compressed list.

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

1) When a context is being initialized, a complete representation of the compressed list of options is transmitted. All options that have any content are present in the compressed list of items sent by the compressor.

Then, once the context has been initialized:

2) When the structure AND the content of the list are unchanged, no information about the list is sent in compressed headers.

3) When the structure of the list is constant, and when only the content defined within the irregular format for one or more options is changed, no information about the list needs to be sent in compressed base headers; the irregular content is sent as part of the irregular chain, as described in Section 6.3.6.

4) When the structure of the list changes, a compressed list is sent in the compressed base header, including a representation of its structure and order. Content defined within the irregular format of an option can still be sent as part of the irregular chain (as described in Section 6.3.6), provided that the item content is not part of the compressed list.

6.3.2. 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 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 successfully verified using the CRC. The decompressor retrieves the item from the table whenever an index without an accompanying item is received.

If an index without an accompanying item is received and the decompressor does not have any context for this index, the header MUST be discarded and a NACK SHOULD be sent.

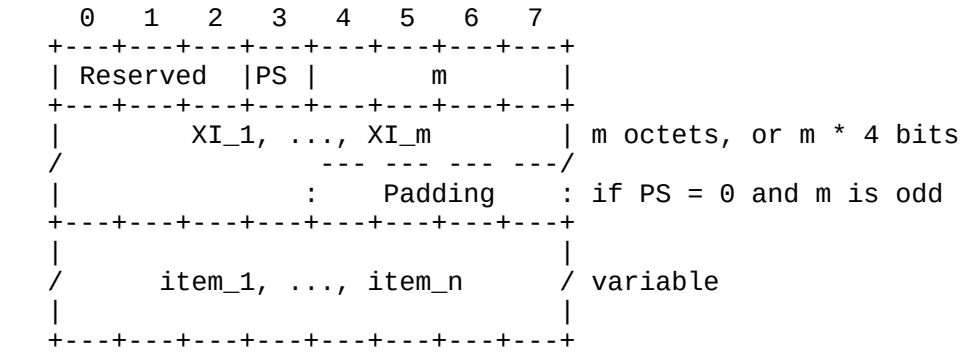
6.3.3. Encoding of Compressed Lists

Each item present in a compressed list is represented by:

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

Decompression of an item will fail if the presence bit is not set and the decompressor has no entry in the context for that item.

A compressed list of TCP options uses the following encoding:



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

PS: Indicates size of XI fields:

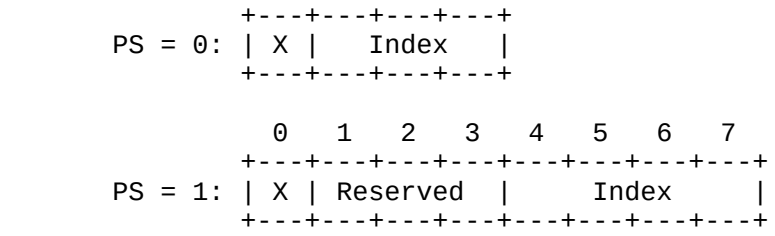
PS = 0 indicates 4-bit XI fields;

PS = 1 indicates 8-bit XI fields.

m: Number of XI item(s) in the compressed list.

XI_1, ..., XI_m: m XI items. Each XI represents one TCP option in the uncompressed packet, in the same order as they appear in the uncompressed packet.

The format of an XI item is as follows:



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

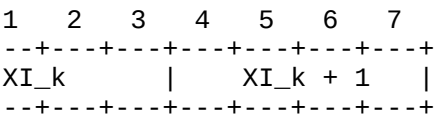
X = 1 indicates that the item corresponding to the Index is sent in the item_1, ..., item_n list;

X = 0 indicates that the item corresponding to the Index is not sent and is instead included in the irregular chain.

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

Index: An index into the item table. See Section 6.3.4.

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. The format of the entries in the item list is described in the table in Section 6.3. The compressed format(s) suffixed by "_list_item" in the encoding methods defines the item inside the compressed item list.

6.3.4. Item Table Mappings

The item table for TCP options list compression is limited to 16 different items, since it is unlikely that any packet flow will contain a larger number of unique options.

The mapping between the TCP option type and table indexes are listed in the table below:

Option name	Table index
NOP	0
EOL	1
MSS	2
WINDOW SCALE	3
TIMESTAMP	4
SACK-PERMITTED	5
SACK	6
Generic options	7-15

Some TCP options are used more frequently than others. To simplify their compression, a part of the item table is reserved for these option types, as shown on the table above. Both the compressor and the decompressor MUST use these mappings between item and indexes to (de)compress TCP options when using list compression.

It is expected that the option types for which an index is reserved in the item table will only appear once in a list. However, if an option type is detected twice in the same options list and if both options have a different content, the compressor should compress the second occurrence of the option type by mapping it to a generic compressed option. Otherwise, if the options have the exact same content, the compressor can still use the same table index for both.

The NOP option

The NOP option can appear more than once in the list. However, since its value is always the same, no context information needs to be transmitted. Multiple NOP options can thus be mapped to the same index. Since the NOP option does not have any content when compressed as a "_list_item", it will never be present in the item list. For consistency, the compressor should still establish an entry in the list by setting the presence bit, as done for the other type of options.

List compression always preserves the original order of each item in the decompressed list, whether or not the item is present in the compressed "_list_item" or if multiple items of the same type can be mapped to the same index, as for the NOP option.

The EOL option

The size of the compressed format for the EOL option can be larger than one octet, and it is defined so that it includes the option padding. This is because the EOL should terminate the parsing of the options, but it can also be followed by padding octets that all have the value zero.

The Generic option

The Generic option can be used to compress any type of TCP option that does not have a reserved index in the item table.

6.3.5. Compressed Lists in Dynamic Chain

A compressed list for TCP options that is part of the dynamic chain (e.g., in IR or IR-DYN packets) must have all its list items present, i.e., all X-bits in the XI list MUST be set.

6.3.6. Irregular Chain Items for TCP Options

The "_list_item" represents the option inside the compressed item list, and the "_irregular" format is used for the option fields that are expected to change with each packet. When an item of the specified type is present in the current context, these irregular fields are present in each compressed packet, as part of the irregular chain. Since many of the TCP option types are not expected to change for the duration of a flow, many of the "_irregular" formats are empty.

The irregular chain for TCP options is structured analogously to the structure of the TCP options in the uncompressed packet. If a compressed list is present in the compressed packet, then the irregular chain for TCP options must not contain irregular items for the list items that are transmitted inside the compressed list (i.e., items in the list that have the X-bit set in its XI). The items that are not present in the compressed list, but are present in the uncompressed list, must have their respective irregular items present in the irregular chain.

6.3.7. Replication of TCP Options

The entire table of TCP options items is always replicated when using the IR-CR packet. In the IR-CR packet, the list of options for the new flow is also transmitted as a compressed list in the IR-CR packet.

6.4. Profile-Specific Encoding Methods

This section defines encoding methods that are specific to this profile. These methods are used in the formal definition of the packet formats in Section 8.

6.4.1. inferred_ip_v4_header_checksum

This encoding method compresses the Header Checksum field of the IPv4 header. This checksum is defined in [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. When almost all IP header information is compressed away, and when decompression is verified by a CRC computed over the original header for every compressed packet, there is no point in having this additional checksum; instead, it can be recomputed at the decompressor side.

The "inferred_ip_v4_header_checksum" encoding method thus compresses the IPv4 header checksum down to a size of zero bits. Using this encoding method, the decompressor infers the value of this field using the computation above.

This encoding method implicitly assumes that the compressor will not process a corrupted header; otherwise, it cannot guarantee that the checksum as recomputed by the decompressor will be bitwise identical to its original value before compression.

6.4.2. inferred_mine_header_checksum

This encoding method compresses the minimal encapsulation header checksum. This checksum is defined in [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 zero. 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 and the assumptions for inferring this checksum are similar to the ones explained above in Section 6.4.1.

6.4.3. 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 [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 Total Length field down to a size of zero bits. Using this encoding method, the decompressor infers the value of this field by counting in octets the length of the entire packet after decompression.

6.4.4. inferred_ip_v6_length

This encoding method compresses the Payload Length field of the IPv6 header. This length field is defined in [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. Using this encoding method, the decompressor infers the value of this field by counting in octets the length of the entire packet after decompression.

6.4.5. inferred_offset

This encoding method compresses the data offset field of the TCP header.

The "inferred_offset" encoding method is used on the Data Offset field of the TCP header. This field is defined in [RFC0793] as:

> Data Offset: 4 bits
>
> The number of 32-bit words in the TCP header. This indicates where the data begins. The TCP header (even one including options) is an integral number of 32 bits long.

The "inferred_offset" encoding method compresses the Data Offset field of the TCP header down to a size of zero bits. Using this encoding method, the decompressor infers the value of this field by first decompressing the TCP options list, and by then setting:

```
data offset = (options length / 4) + 5
```

The equation above uses integer arithmetic.

6.4.6. baseheader_extension_headers

In CO packets (see Section 7.3), the innermost IP header and the TCP header are combined to create a compressed base header. In some cases, the IP header will have a number of extension headers between itself and the TCP header.

To remain formally correct, the base header must define some representation of these extension headers, which is what this encoding method is used for. This encoding method skips over all the extension headers and does not encode any of the fields. Changed fields in these headers are encoded in the irregular chain.

6.4.7. baseheader_outer_headers

This encoding method, as well as the baseheader_extension_headers encoding method described above, is needed for the specification to remain formally correct. It is used in CO packets (see Section 7.3) to describe tunneling IP headers and their respective extension headers (i.e., all headers located before the innermost IP header).

This encoding method skips over all the fields in these headers and does not perform any encoding. Changed fields in outer headers are instead handled by the irregular chain.

6.4.8. Scaled Encoding of Fields

Some header fields will exhibit a change pattern where the field increases by a constant value or by multiples of the same value.

Examples of fields that may have this behavior are the TCP Sequence Number and the TCP Acknowledgment Number. For such fields, ROHC-TCP provides the means to downscale the field value before applying LSB encoding, which allows the compressor to transmit fewer bits.

To be able to use scaled encoding, the field is required to fulfill the following equation:

```
unscaled_value = scaling_factor * scaled_value + residue
```

To use the scaled encoding, the compressor must be confident that the decompressor has established values for the "residue" and the "scaling_factor", so that it can correctly decompress the field when only an LSB-encoded "scaled_value" is present in the compressed packet.

Once the compressor is confident that the value of the scaling_factor and the value of the residue have been established in the decompressor, the compressor may send compressed packets using the scaled representation of the field. The compressor MUST NOT use scaled encoding with the value of the scaling_factor set to zero.

If the compressor detects that the value of the residue has changed, or if the compressor uses a different value for the scaling factor, it MUST NOT use scaled encoding until it is confident that the decompressor has received the new value(s) of these fields.

When the unscaled value of the field wraps around, the value of the residue is likely to change, even if the scaling_factor remains constant. In such a case, the compressor must act in the same way as for any other change in the residue.

The following subsections describe how the scaled encoding is applied to specific fields in ROHC-TCP, in particular, how the scaling_factor and residue values are established for the different fields.

6.4.8.1. Scaled TCP Sequence Number Encoding

For some TCP flows, such as data transfers, the payload size will be constant over periods of time. For such flows, the TCP Sequence Number is bound to increase by multiples of the payload size between packets, which means that this field can be a suitable target for scaled encoding. When using this encoding, the payload size will be used as the scaling factor (i.e., as the value for scaling_factor) of this encoding. This means that the scaling factor does not need to be explicitly transmitted, but is instead inferred from the length of the payload in the compressed packet.

Establishing scaling_factor:

> The scaling factor is established by sending unscaled TCP Sequence Number bits, so that the decompressor can infer the scaling_factor from the payload size.

Establishing residue:

> The residue is established identically as the scaling_factor, i.e., by sending unscaled TCP Sequence Number bits.

A detailed specification of how the TCP Sequence Number uses the scaled encoding can be found in the definitions of the packet formats, in Section 8.2.

6.4.8.2. Scaled Acknowledgment Number Encoding

Similar to the pattern exhibited by the TCP Sequence Number, the expected increase in the TCP Acknowledgment Number is often constant and is therefore suitable for scaled encoding.

For the TCP Acknowledgment Number, the scaling factor depends on the size of packets flowing in the opposite direction; this information might not be available to the compressor/decompressor pair. For this reason, ROHC-TCP uses an explicitly transmitted scaling factor to compress the TCP Acknowledgment Number.

Establishing scaling_factor:

The scaling factor is established by explicitly transmitting the value of the scaling factor (called ack_stride in the formal notation in Section 8.2) to the decompressor, using one of the packet types that can carry this information.

Establishing residue:

The scaling residue is established by sending unscaled TCP Acknowledgment Number bits, so that the decompressor can infer its value from the unscaled value and the scaling factor (ack_stride).

A detailed specification of how the TCP Acknowledgment Number uses the scaled encoding can be found in the definitions of the packet formats, in Section 8.2.

The compressor MAY use the scaled acknowledgment number encoding; what value it will use as the scaling factor is up to the compressor implementation. In the case where there is a co-located decompressor processing packets of the same TCP flow in the opposite direction, the scaling factor for the sequence number used for that flow can be used by the compressor to determine a suitable scaling factor for the TCP Acknowledgment number for this flow.

6.5. Encoding Methods with External Parameters

A number of encoding methods in Section 8.2 have one or more arguments for which the derivation of the parameter's value is outside the scope of the ROHC-FN specification of the header formats. This section lists the encoding methods together with a definition of each of their parameters.

- ipv6(is_innermost, ttl_irregular_chain_flag, ip_inner_ecn):

  is_innermost: This Boolean flag is set to true when processing the innermost IP header; otherwise, it is set to false.

  ttl_irregular_chain_flag: This parameter must be set to the value that was used for the corresponding "ttl_irregular_chain_flag" parameter of the "co_baseheader" encoding method (as defined below) when extracting the irregular chain for a compressed header; otherwise, it is set to zero and ignored for other types of chains.

  ip_inner_ecn: This parameter is bound by the encoding method; therefore, it should be undefined when calling this encoding method. This value is then used to bind the corresponding

parameter in the "tcp" encoding method, as its value is needed when processing the irregular chain for TCP. See the definition of the "ip_inner_ecn" parameter for the "tcp" encoding method below.

- ipv4(is_innermost, ttl_irregular_chain_flag, ip_inner_ecn, ip_id_behavior_value):

  See definition of arguments for "ipv6" above.

  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 8.2.

- tcp_opt_eol(nbits):

  nbits: This parameter is set to the length of the padding data located after the EOL option type octet to the end of the TCP options in the uncompressed header.

- tcp_opt_sack(ack_value):

  ack_value: Set to the value of the Acknowledgment Number field of the TCP header.

- tcp(payload_size, ack_stride_value, ip_inner_ecn):

  payload_size: Set to the length (in octets) of the payload following the TCP header.

  ack_stride_value: This parameter is the scaling factor used when scaling the TCP Acknowledgment Number. Its value is set by the compressor implementation. See Section 6.4.8.2 for recommendations on how to set this value.

  ip_inner_ecn: This parameter binds with the value given to the corresponding "ip_inner_ecn" parameter by the "ipv4" or the "ipv6" encoding method when processing the innermost IP header of this packet. See also the definition of the "ip_inner_ecn" parameter to the "ipv6" and "ipv4" encoding method above.

- co_baseheader(payload_size, ack_stride_value, ttl_irregular_chain_flag, ip_id_behavior_value):

  payload_size: Set to the length (in octets) of the payload following the TCP header.

ack_stride_value: This parameter is the scaling factor used when scaling the TCP Acknowledgment Number. Its value is set by the compressor implementation. See Section 6.4.8.2 for recommendations on how to set this value.

ttl_irregular_chain_flag: This parameter is set to one if the TTL/Hop Limit of an outer header has changed compared to its reference in the context; otherwise, it is set to zero. The value used for this parameter is also used for the "ttl_irregular_chain_flag" argument for the "ipv4" and "ipv6" encoding methods when processing the irregular chain, as defined above for the "ipv6" and "ipv4" encoding methods.

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 8.2.

7. Packet Types (Normative)

ROHC-TCP uses three different packet types: the Initialization and Refresh (IR) packet type, the Context Replication (IR-CR) packet type, and the Compressed (CO) packet type.

Each packet type defines a number of packet formats: two packet formats are defined for the IR type, one packet format is defined for the IR-CR type, and two sets of eight base header formats are defined for the CO type with one additional format that is common to both sets.

The profile identifier for ROHC-TCP is 0x0006.

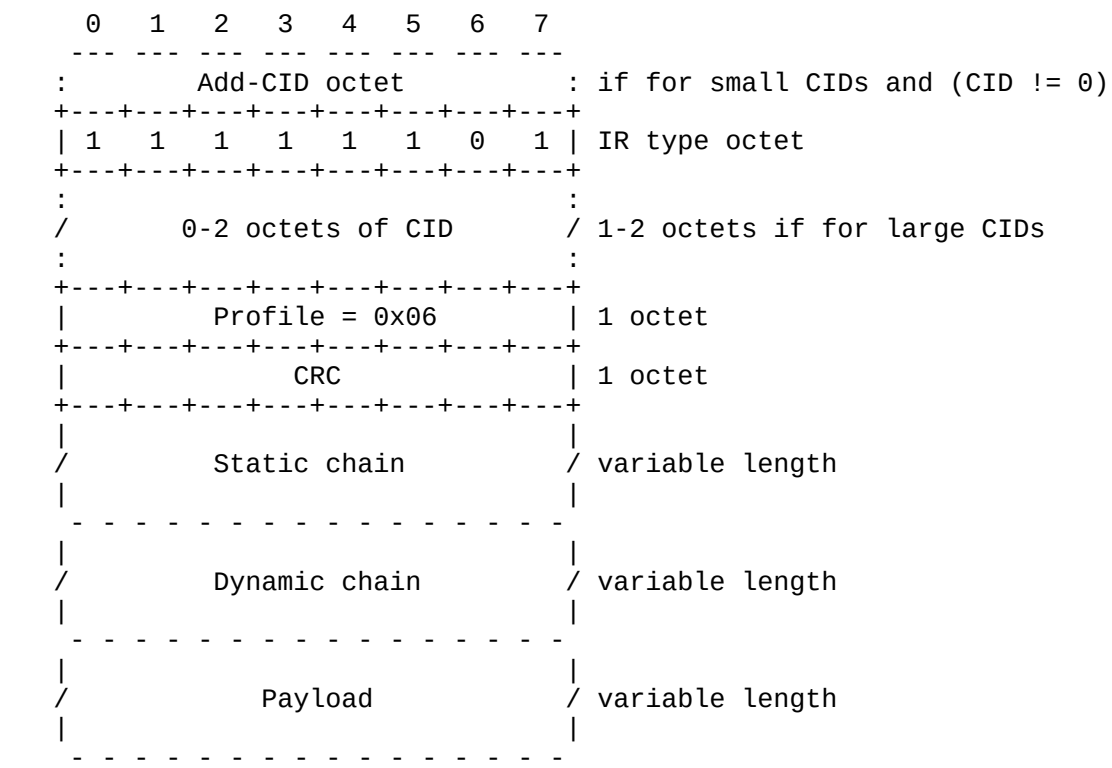
7.1. Initialization and Refresh (IR) Packets

ROHC-TCP uses the basic structure of the ROHC IR and IR-DYN packets as defined in [RFC5795] (Sections 5.2.2.1 and 5.2.2.2, respectively).

Packet type: IR

This packet type communicates the static part and the dynamic part of the context.

For the ROHC-TCP IR packet, the value of the x bit MUST be set to one. It has the following format, which corresponds to the "Header" and "Payload" fields described in Section 5.2.1 of [RFC5795]:



CRC: 8-bit CRC, computed according to Section 5.3.1.1 of [RFC5795]. The CRC covers the entire IR header, thus excluding payload, padding, and feedback, if any.

Static chain: See Section 6.2.

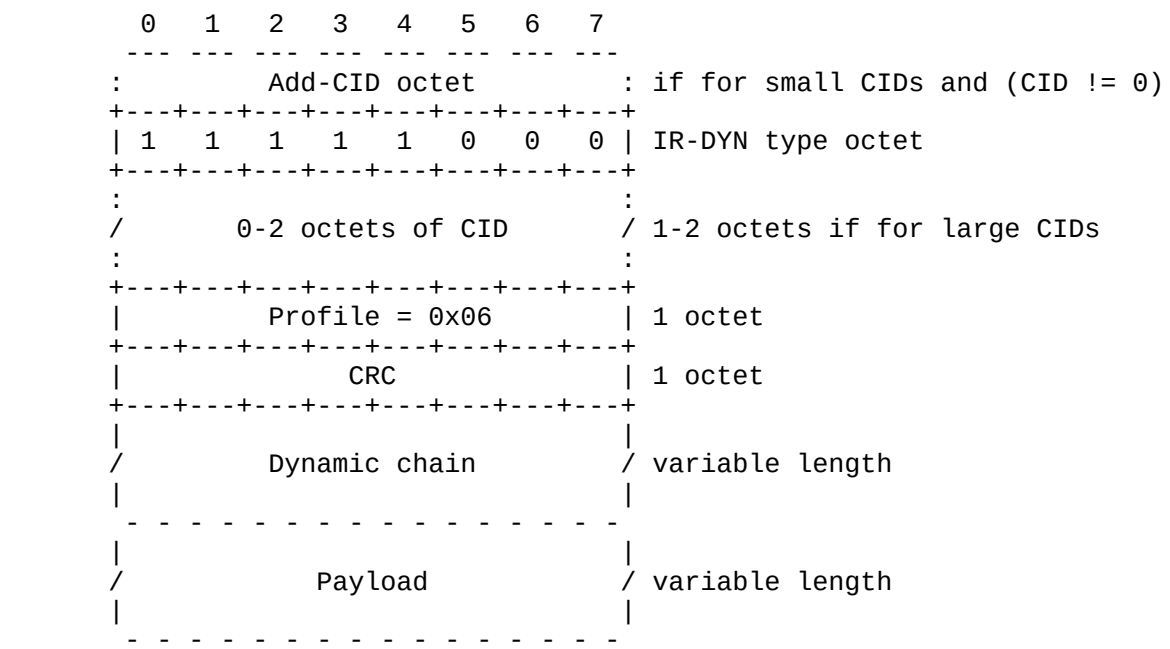
Dynamic chain: See Section 6.2.

Payload: The payload of the corresponding original packet, if any. The payload consists of all data after the last octet of the TCP header to the end of the uncompressed packet. The presence of a payload is inferred from the packet length.

Packet type: IR-DYN

This packet type communicates the dynamic part of the context.

The ROHC-TCP IR-DYN packet has the following format, which corresponds to the "Header" and "Payload" fields described in Section 5.2.1 of [RFC5795]:



CRC: 8-bit CRC, computed according to Section 5.3.1.1 of [RFC5795]. The CRC covers the entire IR-DYN header, thus excluding payload, padding, and feedback, if any.

Dynamic chain: See Section 6.2.

Payload: The payload of the corresponding original packet, if any. The payload consists of all data after the last octet of the TCP header to end of the uncompressed packet. The presence of a payload is inferred from the packet length.

7.2. Context Replication (IR-CR) Packets

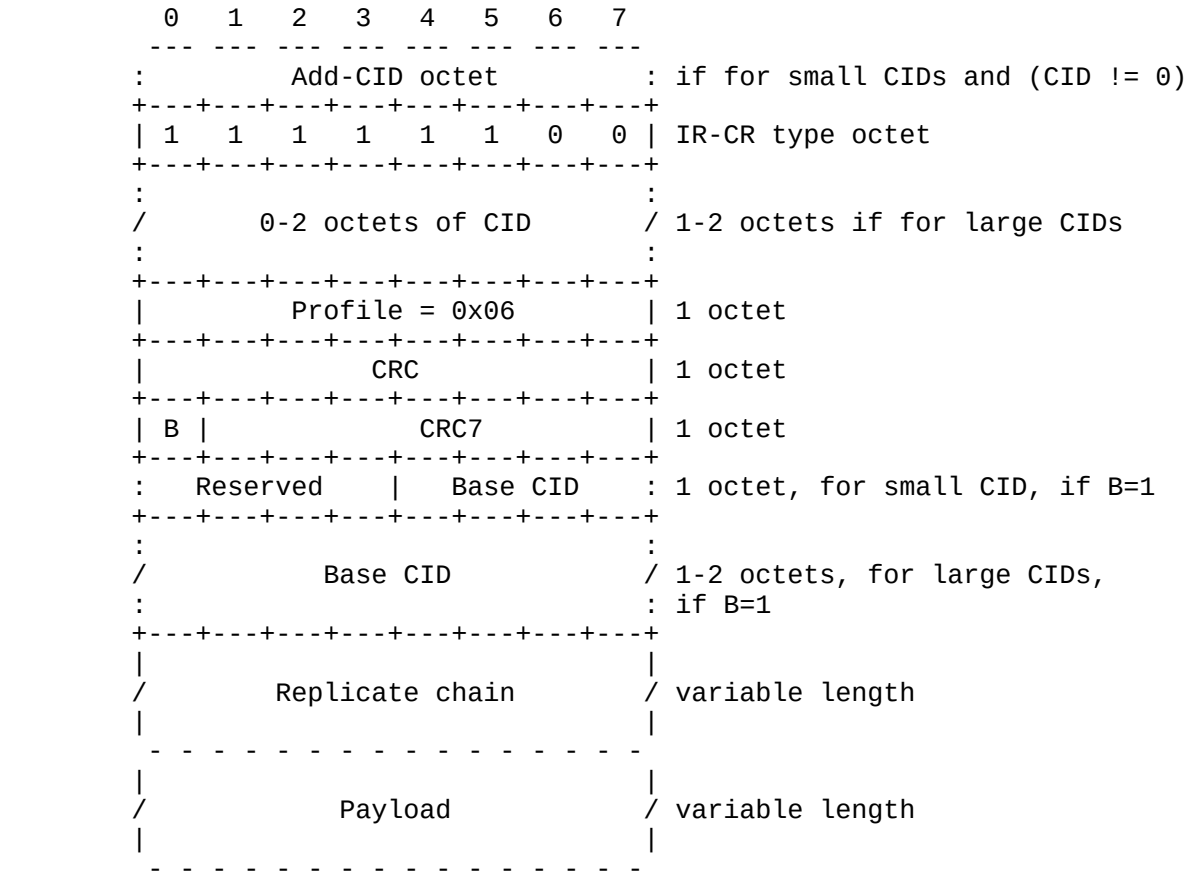
Context replication requires a dedicated IR packet format that uniquely identifies the IR-CR packet for the ROHC-TCP profile. This section defines the profile-specific part of the IR-CR packet [RFC4164].

Packet type: IR-CR

This packet type communicates a reference to a base context along with the static and dynamic parts of the replicated context that differs from the base context.

The ROHC-TCP IR-CR packet follows the general format of the ROHC IR-CR packet, as defined in [RFC4164], Section 3.5.2. With consideration to the extensibility of the IR packet type defined in [RFC5795], the ROHC-TCP profile supports context replication through the profile-specific part of the IR packet. This is achieved using the bit (x) left in the IR header for "Profile specific information". For ROHC-TCP, this bit is defined as a flag indicating whether this packet is an IR packet or an IR-CR packet. For the ROHC-TCP IR-CR packet, the value of the x bit MUST be set to zero.

The ROHC-TCP IR-CR has the following format, which corresponds to the "Header" and "Payload" fields described in Section 5.2.1 of [RFC5795]:



B: B = 1 indicates that the Base CID field is present.

CRC: This CRC covers the entire IR-CR header, thus excluding payload, padding, and feedback, if any. This 8-bit CRC is calculated according to Section 5.3.1.1 of [RFC5795].

CRC7: The CRC over the original, uncompressed, header. Calculated according to Section 3.5.1.1 of [RFC4164].

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

Base CID: CID of base context. Encoded according to [RFC4164], Section 3.5.3.

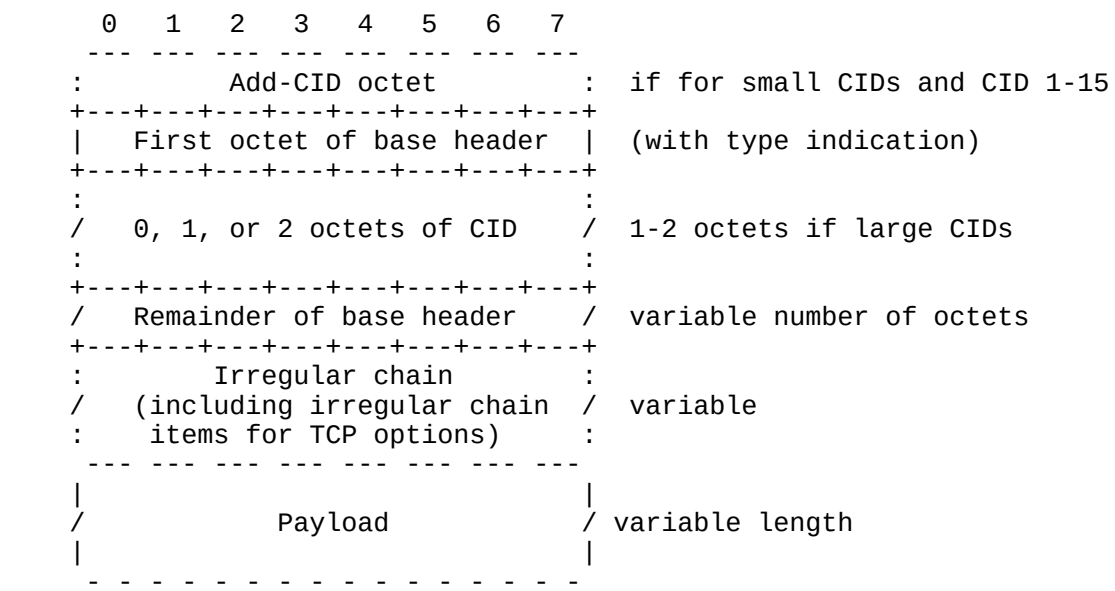
Replicate chain: See Section 6.2.

Payload: The payload of the corresponding original packet, if any. The presence of a payload is inferred from the packet length.

7.3. Compressed (CO) Packets

The ROHC-TCP CO packets communicate irregularities in the packet header. All CO packets carry a CRC and can update the context.

The general format for a compressed TCP header is as follows, which corresponds to the "Header" and "Payload" fields described in Section 5.2.1 of [RFC5795]:



Base header: The complete set of base headers is defined in Section 8.

Irregular chain: See Sections 6.2 and 6.3.6.

Payload: The payload of the corresponding original packet, if any. The presence of a payload is inferred from the packet length.

8. Header Formats (Normative)

This section describes the set of compressed TCP/IP packet formats. The normative description of the packet formats is given using the formal notation for ROHC profiles defined in [RFC4997]. The formal description of the packet formats specifies all of the information needed to compress and decompress a header relative to the context.

In particular, the notation provides a list of all the fields present in the uncompressed and compressed TCP/IP headers, and defines how to map from each uncompressed packet to its compressed equivalent and vice versa.

8.1. Design Rationale for Compressed Base Headers

The compressed header formats are defined as two separate sets: one set for the packets where the innermost IP header contains a sequential IP-ID (either network byte order or byte swapped), and one set for the packets without sequential IP-ID (either random, zero, or no IP-ID).

These two sets of header formats are referred to as the "sequential" and the "random" set of header formats, respectively.

In addition, there is one compressed format that is common to both sets of header formats and that can thus be used regardless of the type of IP-ID behavior. This format can transmit rarely changing fields and also send the frequently changing fields coded in variable lengths. It can also change the value of control fields such as IP-ID behavior and ECN behavior.

All compressed base headers contain a 3-bit CRC, unless they update control fields such as "ip_id_behavior" or "ecn_used" that affect the interpretation of subsequent headers. Headers that can modify these control fields carry a 7-bit CRC instead.

When discussing LSB-encoded fields below, "p" equals the "offset_param" and "k" equals the "num_lsbs_param" in [RFC4997].

The encoding methods used in the compressed base headers are based on the following design criteria:

- MSN

  Since the MSN is a number generated by the compressor, it only needs to be large enough to ensure robust operation and to accommodate a small amount of reordering [RFC4163]. Therefore, each compressed base header has an MSN field that is LSB-encoded with k=4 and p=4 to handle a reordering depth of up to 4 packets. Additional guidance to improve robustness when reordering is possible can be found in [RFC4224].

- TCP Sequence Number

  ROHC-TCP has the capability to handle bulk data transfers efficiently, for which the sequence number is expected to increase by about 1460 octets (which can be represented by 11 bits). For the compressed base headers to handle retransmissions (i.e., negative delta to the sequence number),

the LSB interpretation interval has to handle negative offsets about as large as positive offsets, which means that one more bit is needed.

Also, for ROHC-TCP to be robust to losses, two additional bits are added to the LSB encoding of the sequence number. This means that the base headers should contain at least 14 bits of LSB-encoded sequence number when present. According to the logic above, the LSB offset value is set to be as large as the positive offset, i.e., p = 2^(k-1)-1.

- TCP Acknowledgment Number

  The design criterion for the acknowledgment number is similar to that of the TCP Sequence Number. However, often only every other data packet is acknowledged, which means that the expected delta value is twice as large as for sequence numbers.

  Therefore, at least 15 bits of acknowledgment number should be used in compressed base headers. Since the acknowledgment number is expected to constantly increase, and the only exception to this is packet reordering (either on the ROHC channel [RFC3759] or prior to the compression point), the negative offset for LSB encoding is set to be 1/4 of the total interval, i.e., p = 2^(k-2)-1.

- TCP Window

  The TCP Window field is expected to increase in increments of similar size as the TCP Sequence Number; therefore, the design criterion for the TCP window is to send at least 14 bits when used.

- IP-ID

  For the "sequential" set of packet formats, all the compressed base headers contain LSB-encoded IP-ID offset bits, where the offset is the difference between the value of the MSN field and the value of the IP-ID field. The requirement is that at least 3 bits of IP-ID should always be present, but it is preferable to use 4 to 7 bits. When k=3 then p=1, and if k>3 then p=3 since the offset is expected to increase most of the time.

Each set of header formats contains eight different compressed base headers. The reason for having this large number of header formats is that the TCP Sequence Number, TCP Acknowledgment Number, and TCP Window are frequently changing in a non-linear pattern.

The design of the header formats is derived from the field behavior analysis found in [RFC4413].

All of the compressed base headers transmit LSB-encoded MSN bits, the TCP Push flag, and a CRC, and in addition to this, all the base headers in the sequential packet format set contain LSB-encoded IP-ID bits.

The following header formats exist in both the sequential and random packet format sets:

- Format 1: This header format carries changes to the TCP Sequence Number and is expected to be used on the downstream of a data transfer.

- Format 2: This header format carries the TCP Sequence Number in scaled form and is expected to be useful for the downstream of a data transfer where the payload size is constant for multiple packets.

- Format 3: This header format carries changes in the TCP Acknowledgment Number and is expected to be useful for the acknowledgment direction of a data transfer.

- Format 4: This header format is similar to format 3, but carries a scaled TCP Acknowledgment Number.

- Format 5: This header format carries both the TCP Sequence Number and the TCP Acknowledgment Number and is expected to be useful for flows that send data in both directions.

- Format 6: This header format is similar to format 5, but carries the TCP Sequence Number in scaled form, when the payload size is static for certain intervals in a data flow.

- Format 7: This header format carries changes to both the TCP Acknowledgment Number and the TCP Window and is expected to be useful for the acknowledgment flows of data connections.

- Format 8: This header format is used to convey changes to some of the more seldom changing fields in the TCP flow, such as ECN behavior, RST/SYN/FIN flags, the TTL/Hop Limit, and the TCP options list. This format carries a 7-bit CRC, since it can change the structure of the contents of the irregular chain for subsequent packets. Note that this can be seen as a reduced form of the common packet format.

   o Common header format: The common header format can be used for all kinds of IP-ID behavior and should be useful when some of the more rarely changing fields in the IP or TCP header change. Since this header format can update control fields that decide how the decompressor interprets packets, it carries a 7-bit CRC to reduce the probability of context corruption. This header can basically convey changes to any of the dynamic fields in the IP and TCP headers, and it uses a large set of flags to provide information about which fields are present in the header format.

8.2. Formal Definition of Header Formats

```
// NOTE: The irregular, static, and dynamic chains (see Section 6.2)
// 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_SEQUENTIAL = 0;
IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED = 1;
IP_ID_BEHAVIOR_RANDOM = 2;
IP_ID_BEHAVIOR_ZERO = 3;

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

CONTROL {
  ecn_used            [ 1 ];
  msn                 [ 16 ];
  // ip_id fields are for innermost IP header only
  ip_id_offset                              [ 16 ];
  ip_id_behavior_innermost                  [  2 ];
  // ACK-related
  ack_stride                                [ 32 ];
  ack_number_scaled                         [ 32 ];
  ack_number_residue                        [ 32 ];
  seq_number_scaled                         [ 32 ];
  seq_number_residue                        [ 32 ];
}
```

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

list_tcp_options               "defined in Section 6.3.3";
inferred_ip_v4_header_checksum "defined in Section 6.4.1";
inferred_mine_header_checksum  "defined in Section 6.4.2";
inferred_ip_v4_length          "defined in Section 6.4.3";
inferred_ip_v6_length          "defined in Section 6.4.4";
inferred_offset                "defined in Section 6.4.5";
baseheader_extension_headers   "defined in Section 6.4.6";
baseheader_outer_headers       "defined in Section 6.4.7";

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

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

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

  COMPRESSED static_enc {
    field =:= static [ 0 ];
    ENFORCE(flag == 0);
  }
}

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

  COMPRESSED non_zero {
    field =:= irregular(width) [ width ];
    ENFORCE(flag == 0);
  }

  COMPRESSED zero {
    field =:= uncompressed_value(width, 0) [ 0 ];
    ENFORCE(flag == 1);
```

```
  }
}

variable_length_32_enc(flag)
{
  UNCOMPRESSED {
    field [ 32 ];
  }

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

  COMPRESSED lsb_8_bit {
    field =:= lsb(8, 63) [ 8 ];
    ENFORCE(flag == 1);
  }

  COMPRESSED lsb_16_bit {
    field =:= lsb(16, 16383) [ 16 ];
    ENFORCE(flag == 2);
  }

  COMPRESSED irreg_32_bit {
    field =:= irregular(32) [ 32 ];
    ENFORCE(flag == 3);
  }
}

optional32(flag)
{
  UNCOMPRESSED {
    item [ 0, 32 ];
  }

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

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

lsb_7_or_31
```

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

  COMPRESSED lsb_7 {
    discriminator =:= '0'       [ 1 ];
    item          =:= lsb(7, 8) [ 7 ];
  }

  COMPRESSED lsb_31 {
    discriminator =:= '1'          [ 1 ];
    item          =:= lsb(31, 256) [ 31 ];
  }
}

opt_lsb_7_or_31(flag)
{
  UNCOMPRESSED {
    item [ 0, 32 ];
  }

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

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

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 ];
  }
}

one_bit_choice
{
  UNCOMPRESSED {
    field [ 1 ];
  }

  COMPRESSED zero {
    field [ 1 ];
    ENFORCE(field.UVALUE == 0);
  }

  COMPRESSED nonzero {
    field [ 1 ];
    ENFORCE(field.UVALUE == 1);
  }
}


// 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_0_replicate {
    discriminator =:= '00000000' [ 8 ];
  }

  COMPRESSED dest_opt_1_replicate {
    discriminator =:= '10000000'                    [ 8 ];
    length        =:= irregular(8)                  [ 8 ];
    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_0_replicate {
    discriminator =:= '00000000' [ 8 ];
  }

  COMPRESSED hop_opt_1_replicate {
    discriminator =:= '10000000'                    [ 8 ];
    length        =:= irregular(8)                  [ 8 ];
    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_0_replicate {
    discriminator =:= '00000000' [ 8 ];
  }

  COMPRESSED rout_opt_0_replicate {
    discriminator =:= '10000000'                  [ 8 ];
    length        =:= irregular(8)                [ 8 ];
    value         =:=
      irregular(length.UVALUE*64+48) [ length.UVALUE * 64 + 48 ];
  }
  COMPRESSED rout_opt_irregular {
  }
}

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

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

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

  COMPRESSED not_present {
```

```
    value     =:= compressed_value(0, 0) [ 0 ];
    reserved1 =:= compressed_value(0, 0) [ 0 ];
    ENFORCE(flag_value == 0);
  }
}

gre_proto
{
  UNCOMPRESSED {
    protocol [ 16 ];
  }

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

  COMPRESSED ether_v6 {
    discriminator =:= compressed_value(1, 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      =:= optional32(k_flag.UVALUE) [ 0, 32 ];
  }

  COMPRESSED gre_dynamic {
    checksum_and_res =:=
      optional_checksum(c_flag.UVALUE)             [ 0, 16 ];
    sequence_number  =:= optional32(s_flag.UVALUE) [ 0, 32 ];
  }

  COMPRESSED gre_0_replicate {
    discriminator    =:= '00000000'    [ 8 ];
    checksum_and_res =:=
      optional_checksum(c_flag.UVALUE) [ 0, 16 ];
    sequence_number  =:=
      optional32(s_flag.UVALUE)        [ 0, 8, 32 ];
  }

  COMPRESSED gre_1_replicate {
    discriminator    =:= '10000'                 [ 5 ];
    c_flag           =:= irregular(1)            [ 1 ];
    k_flag           =:= irregular(1)            [ 1 ];
    s_flag           =:= irregular(1)            [ 1 ];
    checksum_and_res =:=
      optional_checksum(c_flag.UVALUE)           [ 0, 16 ];
    key              =:= optional32(k_flag.UVALUE) [ 0, 32 ];
    sequence_number  =:= optional32(s_flag.UVALUE) [ 0, 32 ];
  }

  COMPRESSED gre_irregular {
    checksum_and_res =:=
      optional_checksum(c_flag.UVALUE) [ 0, 16 ];
    sequence_number  =:=
      opt_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    =:= optional32(s_bit.UVALUE) [ 0, 32 ];
  }

  COMPRESSED mine_dynamic {
  }

  COMPRESSED mine_0_replicate {
    discriminator =:= '00000000' [ 8 ];
  }

  COMPRESSED mine_1_replicate {
    discriminator =:= '10000000'               [ 8 ];
    s_bit         =:= irregular(1)             [ 1 ];
    res_bits      =:= irregular(7)             [ 7 ];
    orig_dest     =:= irregular(32)            [ 32 ];
    orig_src      =:= optional32(s_bit.UVALUE) [ 0, 32 ];
  }

  COMPRESSED mine_irregular {
  }
}
```

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

ah
{
  UNCOMPRESSED {
    next_header     [ 8 ];
    length          [ 8 ];
    res_bits        [ 16 ];
    spi             [ 32 ];
    sequence_number [ 32 ];
    icv             [ length.UVALUE*32-32 ];
  }

  DEFAULT {
    next_header     =:= static;
    length          =:= static;
    res_bits        =:= 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 {
    res_bits        =:= irregular(16) [ 16 ];
    sequence_number =:= irregular(32) [ 32 ];
    icv       =:=
      irregular(length.UVALUE*32-32)  [ length.UVALUE*32-32 ];
  }

  COMPRESSED ah_0_replicate {
    discriminator   =:= '00000000'    [ 8 ];
    sequence_number =:= irregular(32) [ 32 ];
    icv             =:=
      irregular(length.UVALUE*32-32)  [ length.UVALUE*32-32 ];
  }

  COMPRESSED ah_1_replicate {
    discriminator   =:= '10000000'    [ 8 ];
    length          =:= irregular(8)  [ 8 ];
    res_bits        =:= irregular(16) [ 16 ];
    spi             =:= irregular(32) [ 32 ];
```

```
    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 ];
  }
}

// The is_innermost flag is true if this is the innermost IP header
// If extracting the irregular chain for a compressed packet:
//   - ttl_irregular_chain_flag must have the same value as it had when
//     processing co_baseheader.
//   - ip_inner_ecn is bound in this encoding method and the value that
//     it gets bound to should be passed to the tcp encoding method
//   For other formats than the irregular chain, these two are ignored
ipv6(is_innermost, ttl_irregular_chain_flag, ip_inner_ecn)
{
  UNCOMPRESSED {
    version         =:= uncompressed_value(4, 6) [ 4 ];
    dscp                                         [ 6 ];
    ip_ecn_flags                                 [ 2 ];
    flow_label                                   [ 20 ];
```

```
    payload_length                                  [ 16 ];
    next_header                                     [ 8 ];
    ttl_hopl                                        [ 8 ];
    src_addr                                        [ 128 ];
    dst_addr                                        [ 128 ];
  }

  DEFAULT {
    dscp           =:= static;
    ip_ecn_flags   =:= 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 ];
    reserved     =:= '00'           [ 2 ];
    flow_label   =:= fl_enc         [ 5, 21 ];
    next_header  =:= irregular(8)   [ 8 ];
    src_addr     =:= irregular(128) [ 128 ];
    dst_addr     =:= irregular(128) [ 128 ];
  }

  COMPRESSED ipv6_dynamic {
    dscp         =:= irregular(6) [ 6 ];
    ip_ecn_flags =:= irregular(2) [ 2 ];
    ttl_hopl     =:= irregular(8) [ 8 ];
  }

  COMPRESSED ipv6_replicate {
    dscp         =:= irregular(6) [ 6 ];
    ip_ecn_flags =:= irregular(2) [ 2 ];
    reserved     =:= '000'        [ 3 ];
    flow_label   =:= fl_enc       [ 5, 21 ];
  }

  COMPRESSED ipv6_outer_without_ttl_irregular {
    dscp         =:= static_or_irreg(ecn_used.UVALUE, 6) [ 0, 6 ];
    ip_ecn_flags =:= static_or_irreg(ecn_used.UVALUE, 2) [ 0, 2 ];
    ENFORCE(ttl_irregular_chain_flag == 0);
    ENFORCE(is_innermost == false);
  }

  COMPRESSED ipv6_outer_with_ttl_irregular {
```

```
    dscp         =:= static_or_irreg(ecn_used.UVALUE, 6) [ 0, 6 ];
    ip_ecn_flags =:= static_or_irreg(ecn_used.UVALUE, 2) [ 0, 2 ];
    ttl_hopl     =:= irregular(8)                        [ 8 ];
    ENFORCE(ttl_irregular_chain_flag == 1);
    ENFORCE(is_innermost == false);
  }

  COMPRESSED ipv6_innermost_irregular {
    ENFORCE(ip_inner_ecn == ip_ecn_flags.UVALUE);
    ENFORCE(is_innermost == true);
  }
}

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

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

// The is_innermost flag is true if this is the innermost IP header
// If extracting the irregular chain for a compressed packet:
//   - ttl_irregular_chain_flag must have the same value as it had when
//     processing co_baseheader.
//   - ip_inner_ecn is bound in this encoding method and the value that
//     it gets bound to should be passed to the tcp encoding method
//   For other formats than the irregular chain, these two are ignored
ipv4(is_innermost, ttl_irregular_chain_flag, ip_inner_ecn,
     ip_id_behavior_value)
{
  UNCOMPRESSED {
    version        =:= uncompressed_value(4, 4)  [ 4 ];
    hdr_length     =:= uncompressed_value(4, 5)  [ 4 ];
    dscp                                         [ 6 ];
    ip_ecn_flags                                 [ 2 ];
    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(reorder_ratio.UVALUE == reorder_ratio_value);
```

```
    ENFORCE(innermost_ip.UVALUE == is_innermost);
    ip_id_behavior_outer [ 2 ];
    innermost_ip [ 1 ];
  }

  DEFAULT {
    dscp           =:= static;
    ip_ecn_flags   =:= 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 ];
    reserved     =:= '0000000'     [ 7 ];
    protocol     =:= irregular(8)  [ 8 ];
    src_addr     =:= irregular(32) [ 32 ];
    dst_addr     =:= irregular(32) [ 32 ];
  }

  COMPRESSED ipv4_innermost_dynamic {
    ENFORCE(is_innermost == 1);
    ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
    reserved       =:= '00000'             [ 5 ];
    df             =:= irregular(1)        [ 1 ];
    ip_id_behavior_innermost =:= irregular(2) [ 2 ];
    dscp           =:= irregular(6)        [ 6 ];
    ip_ecn_flags   =:= irregular(2)        [ 2 ];
    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 ];
    dscp           =:= irregular(6)        [ 6 ];
    ip_ecn_flags   =:= irregular(2)        [ 2 ];
    ttl_hopl       =:= irregular(8)        [ 8 ];
    ip_id          =:=
      ip_id_enc_dyn(ip_id_behavior_outer.UVALUE) [ 0, 16 ];
```

```
  }

  COMPRESSED ipv4_innermost_replicate {
    ENFORCE(is_innermost == 1);
    ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
    reserved       =:= '0000'              [ 4 ];
    ip_id_behavior_innermost =:= irregular(2) [ 2 ];
    ttl_flag       =:= irregular(1)        [ 1 ];
    df             =:= irregular(1)        [ 1 ];
    dscp           =:= irregular(6)        [ 6 ];
    ip_ecn_flags   =:= irregular(2)        [ 2 ];
    ip_id          =:=
      ip_id_enc_dyn(ip_id_behavior_innermost.UVALUE) [ 0, 16 ];
    ttl_hopl     =:=
        static_or_irreg(ttl_flag.UVALUE, 8) [ 0, 8 ];
  }

  COMPRESSED ipv4_outer_replicate {
    ENFORCE(is_innermost == 0);
    ENFORCE(ip_id_behavior_outer.UVALUE == ip_id_behavior_value);
    reserved       =:= '0000'              [ 4 ];
    ip_id_behavior_outer =:= irregular(2)  [ 2 ]
    ttl_flag       =:= irregular(1)        [ 1 ];
    df             =:= irregular(1)        [ 1 ];
    dscp           =:= irregular(6)        [ 6 ];
    ip_ecn_flags   =:= irregular(2)        [ 2 ];
    ip_id          =:=
      ip_id_enc_dyn(ip_id_behavior_outer.UVALUE) [ 0, 16 ];
    ttl_hopl     =:=
        static_or_irreg(ttl_flag.UVALUE, 8) [ 0, 8 ];
  }

  COMPRESSED ipv4_outer_without_ttl_irregular {
    ENFORCE(is_innermost == 0);
    ip_id        =:=
      ip_id_enc_irreg(ip_id_behavior_outer.UVALUE) [ 0, 16 ];
    dscp         =:= static_or_irreg(ecn_used.UVALUE, 6) [ 0, 6 ];
    ip_ecn_flags =:= static_or_irreg(ecn_used.UVALUE, 2) [ 0, 2 ];
    ENFORCE(ttl_irregular_chain_flag == 0);
  }

  COMPRESSED ipv4_outer_with_ttl_irregular {
    ENFORCE(is_innermost == 0);
    ip_id        =:=
      ip_id_enc_irreg(ip_id_behavior_outer.UVALUE)       [ 0, 16 ];
    dscp         =:= static_or_irreg(ecn_used.UVALUE, 6) [ 0, 6 ];
    ip_ecn_flags =:= static_or_irreg(ecn_used.UVALUE, 2) [ 0, 2 ];
    ttl_hopl     =:= irregular(8)                        [ 8 ];
```

```
    ENFORCE(ttl_irregular_chain_flag == 1);
  }

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

////////////////////////////////////////////
// TCP Options
////////////////////////////////////////////

// nbits is bound to the remaining length (in bits) of TCP
// options, including the EOL type byte.
tcp_opt_eol(nbits)
{
  UNCOMPRESSED {
    type     =:= uncompressed_value(8, 0) [ 8 ];
    padding  =:=
      uncompressed_value(nbits-8, 0)      [ nbits-8 ];
  }

  CONTROL {
    pad_len [ 8 ];
  }

  COMPRESSED eol_list_item {
    pad_len =:= compressed_value(8, nbits-8) [ 8 ];
  }

  COMPRESSED eol_irregular {
    pad_len =:= static;
    ENFORCE(nbits-8 == pad_len.UVALUE);
  }
}

tcp_opt_nop
{
  UNCOMPRESSED {
    type =:= uncompressed_value(8, 1) [ 8 ];
  }

  COMPRESSED nop_list_item {
  }
```

```
  COMPRESSED nop_irregular {
  }
}

tcp_opt_mss
{
  UNCOMPRESSED {
    type   =:= uncompressed_value(8, 2) [ 8 ];
    length =:= uncompressed_value(8, 4) [ 8 ];
    mss                                 [ 16 ];
  }

  COMPRESSED mss_list_item {
    mss =:= irregular(16) [ 16 ];
  }

  COMPRESSED mss_irregular {
    mss    =:= static;
  }
}

tcp_opt_wscale
{
  UNCOMPRESSED {
    type   =:= uncompressed_value(8, 3) [ 8 ];
    length =:= uncompressed_value(8, 3) [ 8 ];
    wscale                              [ 8 ];
  }

  COMPRESSED wscale_list_item {
    wscale =:= irregular(8) [ 8 ];
  }

  COMPRESSED wscale_irregular {
    wscale =:= static;
  }
}

ts_lsb
{
  UNCOMPRESSED {
    tsval [ 32 ];
  }

  COMPRESSED tsval_7 {
    discriminator =:= '0'        [ 1 ];
    tsval         =:= lsb(7, -1) [ 7 ];
  }
```

```
  COMPRESSED tsval_14 {
    discriminator =:= '10'        [ 2 ];
    tsval         =:= lsb(14, -1) [ 14 ];
  }

  COMPRESSED tsval_21 {
    discriminator =:= '110'               [ 3 ];
    tsval         =:= lsb(21, 0x00040000) [ 21 ];
  }

  COMPRESSED tsval_29 {
    discriminator =:= '111'               [ 3 ];
    tsval         =:= lsb(29, 0x04000000) [ 29 ];
  }
}

tcp_opt_ts
{
  UNCOMPRESSED {
    type   =:= uncompressed_value(8, 8)  [ 8 ];
    length =:= uncompressed_value(8, 10) [ 8 ];
    tsval                                [ 32 ];
    tsecho                               [ 32 ];
  }
  COMPRESSED tsopt_list_item {
    tsval  =:= irregular(32) [ 32 ];
    tsecho =:= irregular(32) [ 32 ];
  }

  COMPRESSED tsopt_irregular {
    tsval  =:= ts_lsb [ 8, 16, 24, 32 ];
    tsecho =:= ts_lsb [ 8, 16, 24, 32 ];
  }
}
sack_pure_lsb(base)
{
  UNCOMPRESSED {
    sack_field [ 32 ];
  }

  CONTROL {
    ENFORCE(sack_field.CVALUE == (sack_field.UVALUE - base));
  }

  COMPRESSED lsb_15 {
    ENFORCE(sack_field.CVALUE == sack_field.CVALUE <= 0x7fff);
    discriminator =:= '0'           [ 1 ];
    sack_field                      [ 15 ];
```

```
  }

  COMPRESSED lsb_22 {
    ENFORCE(sack_field.CVALUE == sack_field.CVALUE <= 0x3fffff);
    discriminator =:= '10'          [ 2 ];
    sack_field                      [ 22 ];
  }

  COMPRESSED lsb_29 {
    ENFORCE(sack_field.CVALUE == sack_field.CVALUE <= 0x1fffffff);
    discriminator =:= '110'         [ 3 ];
    sack_field                      [ 29 ];
  }

  COMPRESSED full_offset {
    discriminator =:= '11111111'    [ 8 ];
    sack_field                      [ 32 ];
  }
}

sack_block(reference)
{
  UNCOMPRESSED {
    block_start [ 32 ];
    block_end   [ 32 ];
  }

  COMPRESSED {
    block_start =:=
      sack_pure_lsb(reference)          [ 16, 24, 32, 40 ];
    block_end   =:=
      sack_pure_lsb(block_start.UVALUE) [ 16, 24, 32, 40 ];
  }
}

// The value of the parameter is set to the ack_number value
// of the TCP header
tcp_opt_sack(ack_value)
{

  UNCOMPRESSED {
    type    =:= uncompressed_value(8, 5) [ 8 ];
    length                               [ 8 ];
    block_1                              [ 64 ];
    block_2                              [ 0, 64 ];
    block_3                              [ 0, 64 ];
    block_4                              [ 0, 64 ];
  }
```

```
  DEFAULT {
    length  =:= static;
    block_2 =:= uncompressed_value(0, 0);
    block_3 =:= uncompressed_value(0, 0);
    block_4 =:= uncompressed_value(0, 0);
  }

  COMPRESSED sack1_list_item {
    discriminator =:= '00000001';
    block_1       =:= sack_block(ack_value);
    ENFORCE(length.UVALUE == 10);
  }

  COMPRESSED sack2_list_item {
    discriminator =:= '00000010';
    block_1       =:= sack_block(ack_value);
    block_2       =:= sack_block(block_1.UVALUE && 0xFFFFFFFF);
    ENFORCE(length.UVALUE == 18);
  }

COMPRESSED sack3_list_item {
   discriminator =:= '00000011';
   block_1       =:= sack_block(ack_value);
   block_2       =:= sack_block(block_1.UVALUE && 0xFFFFFFFF);
   block_3       =:= sack_block(block_2.UVALUE && 0xFFFFFFFF);
   ENFORCE(length.UVALUE == 26);
 }

 COMPRESSED sack4_list_item {
   discriminator =:= '00000100';
   block_1       =:= sack_block(ack_value);
   block_2       =:= sack_block(block_1.UVALUE && 0xFFFFFFFF);
   block_3       =:= sack_block(block_2.UVALUE && 0xFFFFFFFF);
   block_4       =:= sack_block(block_3.UVALUE && 0xFFFFFFFF);
   ENFORCE(length.UVALUE == 34);
 }

  COMPRESSED sack_unchanged_irregular {
    discriminator =:= '00000000';
    block_1       =:= static;
    block_2       =:= static;
    block_3       =:= static;
    block_4       =:= static;
  }

  COMPRESSED sack1_irregular {
    discriminator =:= '00000001';
    block_1       =:= sack_block(ack_value);
```

```
    ENFORCE(length.UVALUE == 10);
  }

 COMPRESSED sack2_irregular {
   discriminator =:= '00000010';
   block_1       =:= sack_block(ack_value);
   block_2       =:= sack_block(block_1.UVALUE && 0xFFFFFFFF);
   ENFORCE(length.UVALUE == 18);
 }

 COMPRESSED sack3_irregular {
   discriminator =:= '00000011';
   block_1       =:= sack_block(ack_value);
   block_2       =:= sack_block(block_1.UVALUE && 0xFFFFFFFF);
   block_3       =:= sack_block(block_1.UVALUE && 0xFFFFFFFF);
   ENFORCE(length.UVALUE == 26);
 }

 COMPRESSED sack4_irregular {
   discriminator =:= '00000100';
   block_1       =:= sack_block(ack_value);
   block_2       =:= sack_block(block_1.UVALUE && 0xFFFFFFFF);
   block_3       =:= sack_block(block_2.UVALUE && 0xFFFFFFFF);
   block_4       =:= sack_block(block_3.UVALUE && 0xFFFFFFFF);
   ENFORCE(length.UVALUE == 34);
 }
}

tcp_opt_sack_permitted
{
  UNCOMPRESSED {
    type   =:= uncompressed_value(8, 4) [ 8 ];
    length =:= uncompressed_value(8, 2) [ 8 ];
  }

  COMPRESSED sack_permitted_list_item {
  }

  COMPRESSED sack_permitted_irregular {
  }
}

tcp_opt_generic
{
  UNCOMPRESSED {
    type                                  [ 8 ];
    length_msb =:= uncompressed_value(1, 0) [ 1 ];
    length_lsb                            [ 7 ];
```

```
    contents                          [ length_lsb.UVALUE*8-16 ];
  }

  CONTROL {
    option_static [ 1 ];
  }

  DEFAULT {
    type       =:= static;
    length_lsb =:= static;
    contents   =:= static;
  }

  COMPRESSED generic_list_item {
    type          =:= irregular(8)      [ 8 ];
    option_static =:= one_bit_choice    [ 1 ];
    length_lsb    =:= irregular(7)      [ 7 ];
    contents      =:=
      irregular(length_lsb.UVALUE*8-16) [ length_lsb.UVALUE*8-16 ];
  }

  // Used when context of option has option_static set to one
  COMPRESSED generic_static_irregular {
    ENFORCE(option_static.UVALUE == 1);
  }

  // An item that can change, but currently is unchanged
  COMPRESSED generic_stable_irregular {
    discriminator =:= '11111111' [ 8 ];
    ENFORCE(option_static.UVALUE == 0);
  }

  // An item that is assumed to change constantly.
  // Length is not allowed to change here, since a length change is
  // most likely to cause new NOPs or an EOL length change.
  COMPRESSED generic_full_irregular {
    discriminator =:= '00000000'        [ 8 ];
    contents      =:=
      irregular(length_lsb.UVALUE*8-16) [ length_lsb.UVALUE*8-16 ];
    ENFORCE(option_static.UVALUE == 0);
  }
}

tcp_list_presence_enc(presence)
{
  UNCOMPRESSED {
    tcp_options;
  }
```

```
  COMPRESSED list_not_present {
    tcp_options =:= static [ 0 ];
    ENFORCE(presence == 0);
  }

  COMPRESSED list_present {
    tcp_options =:= list_tcp_options [ VARIABLE ];
    ENFORCE(presence == 1);
  }
}

////////////////////////////////////////
// TCP Header
////////////////////////////////////////

port_replicate(flags)
{
  UNCOMPRESSED {
    port [ 16 ];
  }

  COMPRESSED port_static_enc {
    port =:= static [ 0 ];
    ENFORCE(flags == 0b00);
  }

  COMPRESSED port_lsb8 {
    port =:= lsb(8, 64) [ 8 ];
    ENFORCE(flags == 0b01);
  }

  COMPRESSED port_irr_enc {
    port =:= irregular(16) [ 16 ];
    ENFORCE(flags == 0b10);
  }
}

tcp_irreg_ip_ecn(ip_inner_ecn)
{
  UNCOMPRESSED {
    ip_ecn_flags [ 2 ];
  }

  COMPRESSED ecn_present {
    // This field does not exist in the uncompressed header
    // and therefore cannot use uncompressed_value.
    ip_ecn_flags =:=
      compressed_value(2, ip_inner_ecn) [ 2 ];
```

```
    ENFORCE(ecn_used.UVALUE == 1);
  }

  COMPRESSED ecn_not_present {
    ip_ecn_flags =:= static [ 0 ];
    ENFORCE(ecn_used.UVALUE == 0);
  }
}

rsf_index_enc
{
  UNCOMPRESSED {
    rsf_flag [ 3 ];
  }

  COMPRESSED none {
    rsf_idx  =:= '00' [ 2 ];
    rsf_flag =:= uncompressed_value(3, 0x00);
  }

  COMPRESSED rst_only {
    rsf_idx  =:= '01' [ 2 ];
    rsf_flag =:= uncompressed_value(3, 0x04);
  }

  COMPRESSED syn_only {
    rsf_idx  =:= '10' [ 2 ];
    rsf_flag =:= uncompressed_value(3, 0x02);
  }

  COMPRESSED fin_only {
    rsf_idx  =:= '11' [ 2 ];
    rsf_flag =:= uncompressed_value(3, 0x01);
  }
}

optional_2bit_padding(used_flag)
{
  UNCOMPRESSED {
  }

  COMPRESSED used {
    padding =:= compressed_value(2, 0x0) [ 2 ];
    ENFORCE(used_flag == 1);
  }

  COMPRESSED unused {
    padding =:= compressed_value(0, 0x0);
```

```
    ENFORCE(used_flag == 0);
  }
}

// ack_stride_value is the user-selected stride for scaling the
// TCP ack_number
// ip_inner_ecn is the value bound when processing the innermost
// IP header (ipv4 or ipv6 encoding method)
tcp(payload_size, ack_stride_value, ip_inner_ecn)
{
  UNCOMPRESSED {
    src_port      [ 16 ];
    dst_port      [ 16 ];
    seq_number    [ 32 ];
    ack_number    [ 32 ];
    data_offset   [ 4 ];
    tcp_res_flags [ 4 ];
    tcp_ecn_flags [ 2 ];
    urg_flag      [ 1 ];
    ack_flag      [ 1 ];
    psh_flag      [ 1 ];
    rsf_flags     [ 3 ];
    window        [ 16 ];
    checksum      [ 16 ];
    urg_ptr       [ 16 ];
    options       [ (data_offset.UVALUE-5)*32 ];
  }

  CONTROL {
    dummy_field_s =:= field_scaling(payload_size,
      seq_number_scaled.UVALUE, seq_number.UVALUE,
      seq_number_residue.UVALUE) [ 0 ];
    dummy_field_a =:= field_scaling(ack_stride.UVALUE,
      ack_number_scaled.UVALUE, ack_number.UVALUE,
      ack_number_residue.UVALUE) [ 0 ];
    ENFORCE(ack_stride.UVALUE == ack_stride_value);
  }

  INITIAL {
    ack_stride     =:= uncompressed_value(16, 0);
  }

  DEFAULT {
    src_port      =:= static;
    dst_port      =:= static;
    seq_number    =:= static;
    ack_number    =:= static;
    data_offset   =:= inferred_offset;
```

```
    tcp_res_flags =:= static;
    tcp_ecn_flags =:= static;
    urg_flag      =:= static;
    ack_flag      =:= uncompressed_value(1, 1);
    rsf_flags     =:= uncompressed_value(3, 0);
    window        =:= static;
    urg_ptr       =:= static;
    ack_stride    =:= static;
    ack_number_scaled =:= static;
    seq_number_scaled =:= static;
    ack_number_residue =:= static;
    seq_number_residue =:= static;
  }

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

  COMPRESSED tcp_dynamic {
    ecn_used        =:= one_bit_choice          [ 1 ];
    ack_stride_flag =:= irregular(1)            [ 1 ];
    ack_zero        =:= irregular(1)            [ 1 ];
    urp_zero        =:= irregular(1)            [ 1 ];
    tcp_res_flags   =:= irregular(4)            [ 4 ];
    tcp_ecn_flags   =:= irregular(2)            [ 2 ];
    urg_flag        =:= irregular(1)            [ 1 ];
    ack_flag        =:= irregular(1)            [ 1 ];
    psh_flag        =:= irregular(1)            [ 1 ];
    rsf_flags       =:= irregular(3)            [ 3 ];
    msn             =:= irregular(16)           [ 16 ];
    seq_number      =:= irregular(32)           [ 32 ];
    ack_number      =:=
      zero_or_irreg(ack_zero.CVALUE, 32)        [ 0, 32 ];
    window          =:= irregular(16)           [ 16 ];
    checksum        =:= irregular(16)           [ 16 ];
    urg_ptr         =:=
      zero_or_irreg(urp_zero.CVALUE, 16)        [ 0, 16 ];
    ack_stride      =:=
      static_or_irreg(ack_stride_flag.CVALUE, 16) [ 0, 16 ];
    options         =:= list_tcp_options        [ VARIABLE ];
  }

  COMPRESSED tcp_replicate {
    reserved          =:= '0'                     [ 1 ];
    window_presence   =:= irregular(1)            [ 1 ];
    list_present      =:= irregular(1)            [ 1 ];
    src_port_presence =:= irregular(2)            [ 2 ];
```

```
    dst_port_presence =:= irregular(2)             [ 2 ];
    ack_stride_flag   =:= irregular(1)             [ 1 ];
    ack_presence      =:= irregular(1)             [ 1 ];
    urp_presence      =:= irregular(1)             [ 1 ];
    urg_flag          =:= irregular(1)             [ 1 ];
    ack_flag          =:= irregular(1)             [ 1 ];
    psh_flag          =:= irregular(1)             [ 1 ];
    rsf_flags         =:= rsf_index_enc            [ 2 ];
    ecn_used          =:= one_bit_choice           [ 1 ];
    msn               =:= irregular(16)            [ 16 ];
    seq_number        =:= irregular(32)            [ 32 ];
    src_port          =:=
      port_replicate(src_port_presence)      [ 0, 8, 16 ];
    dst_port          =:=
      port_replicate(dst_port_presence)      [ 0, 8, 16 ];
    window            =:=
      static_or_irreg(window_presence, 16)   [ 0, 16 ];
    urg_point         =:=
      static_or_irreg(urp_presence, 16)    [ 0, 16 ];
    ack_number        =:=
      static_or_irreg(ack_presence, 32)    [ 0, 32 ];
    ecn_padding       =:=
      optional_2bit_padding(ecn_used.CVALUE)     [ 0, 2 ];
    tcp_res_flags =:=
      static_or_irreg(ecn_used.CVALUE, 4)        [ 0, 4 ];
    tcp_ecn_flags     =:=
      static_or_irreg(ecn_used.CVALUE, 2)        [ 0, 2 ];
    checksum          =:= irregular(16)            [ 16 ];
    ack_stride        =:=
      static_or_irreg(ack_stride_flag.CVALUE, 16)  [ 0, 16 ];
    options           =:=
      tcp_list_presence_enc(list_present.CVALUE) [ VARIABLE ];
  }

  COMPRESSED tcp_irregular {
    ip_ecn_flags  =:= tcp_irreg_ip_ecn(ip_inner_ecn)  [ 0, 2 ];
    tcp_res_flags =:=
      static_or_irreg(ecn_used.CVALUE, 4)            [ 0, 4 ];
    tcp_ecn_flags =:=
      static_or_irreg(ecn_used.CVALUE, 2)             [ 0, 2 ];
    checksum      =:= irregular(16)                   [ 16 ];
  }
}

////////////////////////////////////////////////
// Encoding methods used in compressed base headers
////////////////////////////////////////////////
```

```
dscp_enc(flag)
{
  UNCOMPRESSED {
    dscp [ 6 ];
  }

  COMPRESSED static_enc {
    dscp =:= static [ 0 ];
    ENFORCE(flag == 0);
  }

  COMPRESSED irreg {
    dscp    =:= irregular(6)           [ 6 ];
    padding =:= compressed_value(2, 0) [ 2 ];
    ENFORCE(flag == 1);
  }
}

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

  CONTROL {
    ip_id_nbo    [ 16 ];
  }

  COMPRESSED nbo {
    ip_id_offset =:= lsb(k, p) [ k ];
    ENFORCE(behavior == IP_ID_BEHAVIOR_SEQUENTIAL);
    ENFORCE(ip_id_offset.UVALUE == ip_id.UVALUE - msn.UVALUE);
  }

  COMPRESSED non_nbo {
    ip_id_offset =:= lsb(k, p) [ k ];
    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);
  }
}

optional_ip_id_lsb(behavior, indicator)
{
  UNCOMPRESSED {
    ip_id [ 16 ];
```

```
  }

  COMPRESSED short {
    ip_id =:= ip_id_lsb(behavior, 8, 3) [ 8 ];
    ENFORCE((behavior == IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (behavior == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    ENFORCE(indicator == 0);
  }

  COMPRESSED long {
    ip_id =:= irregular(16)  [ 16 ];
    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);
  }

  COMPRESSED not_present {
    ENFORCE((behavior == IP_ID_BEHAVIOR_RANDOM) ||
            (behavior == IP_ID_BEHAVIOR_ZERO));
  }
}
dont_fragment(version)
{
  UNCOMPRESSED {
    df [ 1 ];
  }

  COMPRESSED v4 {
    df =:= irregular(1) [ 1 ];
    ENFORCE(version == 4);
  }

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

////////////////////////////////
// Actual start of compressed packet formats
// Important note:
//   The base header is the compressed representation
//   of the innermost IP header AND the TCP header.
////////////////////////////////

// ttl_irregular_chain_flag is set by the user if the TTL/Hop Limit
// of an outer header has changed.  The same value must be passed as
```

```
// an argument to the ipv4/ipv6 encoding methods when extracting
// the irregular chain items.
co_baseheader(payload_size, ack_stride_value,
              ttl_irregular_chain_flag, ip_id_behavior_value)
{
  UNCOMPRESSED v4 {
    outer_headers  =:= baseheader_outer_headers        [ VARIABLE ];
    version        =:= uncompressed_value(4, 4)        [ 4 ];
    header_length  =:= uncompressed_value(4, 5)        [ 4 ];
    dscp                                               [ 6 ];
    ip_ecn_flags                                       [ 2 ];
    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 ];
    checksum                                           [ 16 ];
    src_addr                                           [ 32 ];
    dest_addr                                          [ 32 ];
    extension_headers =:= baseheader_extension_headers [ VARIABLE ];
    src_port                                           [ 16 ];
    dest_port                                          [ 16 ];
    seq_number                                         [ 32 ];
    ack_number                                         [ 32 ];
    data_offset                                        [ 4 ];
    tcp_res_flags                                      [ 4 ];
    tcp_ecn_flags                                      [ 2 ];
    urg_flag                                           [ 1 ];
    ack_flag                                           [ 1 ];
    psh_flag                                           [ 1 ];
    rsf_flags                                          [ 3 ];
    window                                             [ 16 ];
    tcp_checksum                                       [ 16 ];
    urg_ptr                                            [ 16 ];
    options                          [ (data_offset.UVALUE-5)*32 ];
  }

  UNCOMPRESSED v6 {
    ENFORCE(ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_RANDOM);
    outer_headers  =:= baseheader_outer_headers        [ VARIABLE ];
    version =:= uncompressed_value(4, 6)               [ 4 ];
    dscp                                               [ 6 ];
    ip_ecn_flags                                       [ 2 ];
    flow_label                                         [ 20 ];
    payload_length                                     [ 16 ];
```

```
    next_header                                          [ 8 ];
    ttl_hopl                                             [ 8 ];
    src_addr                                             [ 128 ];
    dest_addr                                            [ 128 ];
    extension_headers =:= baseheader_extension_headers [ VARIABLE ];
    src_port                                             [ 16 ];
    dest_port                                            [ 16 ];
    seq_number                                           [ 32 ];
    ack_number                                           [ 32 ];
    data_offset                                          [ 4 ];
    tcp_res_flags                                        [ 4 ];
    tcp_ecn_flags                                        [ 2 ];
    urg_flag                                             [ 1 ];
    ack_flag                                             [ 1 ];
    psh_flag                                             [ 1 ];
    rsf_flags                                            [ 3 ];
    window                                               [ 16 ];
    tcp_checksum                                         [ 16 ];
    urg_ptr                                              [ 16 ];
    options                              [ (data_offset.UVALUE-5)*32 ];
    df    =:= uncompressed_value(0,0)                    [   0 ];
    ip_id =:= uncompressed_value(0,0)                    [   0 ];
  }

  CONTROL {
    dummy_field_s =:= field_scaling(payload_size,
      seq_number_scaled.UVALUE, seq_number.UVALUE,
      seq_number_residue.UVALUE) [ 0 ];
    dummy_field_a =:= field_scaling(ack_stride.UVALUE,
      ack_number_scaled.UVALUE, ack_number.UVALUE,
      ack_number_residue.UVALUE) [ 0 ];
    ENFORCE(ack_stride.UVALUE == ack_stride_value);
    ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
  }

  INITIAL {
    ack_stride     =:= uncompressed_value(16, 0);
  }

  DEFAULT {
    tcp_ecn_flags  =:= static;
    data_offset    =:= inferred_offset;
    tcp_res_flags  =:= static;
    rsf_flags      =:= uncompressed_value(3, 0);
    dest_port      =:= static;
    dscp           =:= static;
    src_port       =:= static;
    urg_flag       =:= uncompressed_value(1, 0);
```

```
    window         =:= static;
    dest_addr      =:= static;
    version        =:= static;
    ttl_hopl       =:= static;
    src_addr       =:= static;
    df             =:= static;
    ack_number     =:= static;
    urg_ptr        =:= static;
    seq_number     =:= static;
    ack_flag       =:= uncompressed_value(1, 1);
    // The default for "options" is case 2) and 3) from
    // the list in Section 6.3.1 (i.e., nothing present in the
    // baseheader itself).
    payload_length =:= inferred_ip_v6_length;
    checksum       =:= inferred_ip_v4_header_checksum;
    length         =:= inferred_ip_v4_length;
    flow_label     =:= static;
    next_header    =:= static;
    ip_ecn_flags   =:= static;
    // The tcp_checksum has no default,
    // it is considered a part of tcp_irregular
    ip_id_behavior_innermost =:= static;
    ecn_used       =:= static;
    ack_stride    =:= static;
    ack_number_scaled =:= static;
    seq_number_scaled =:= static;
    ack_number_residue =:= static;
    seq_number_residue =:= static;

    // Default is to have no TTL in irregular chain
    // Can only be nonzero if co_common is used
    ENFORCE(ttl_irregular_chain_flag == 0);
  }

  ////////////////////////////////////////
  // Common compressed packet format
  ////////////////////////////////////////

  COMPRESSED co_common {
    discriminator        =:= '1111101'               [ 7 ];
    ttl_hopl_outer_flag  =:=
        compressed_value(1, ttl_irregular_chain_flag) [ 1 ];
    ack_flag             =:= irregular(1)            [ 1 ];
    psh_flag             =:= irregular(1)            [ 1 ];
    rsf_flags            =:= rsf_index_enc           [ 2 ];
    msn                  =:= lsb(4, 4)               [ 4 ];
    seq_indicator        =:= irregular(2)            [ 2 ];
    ack_indicator        =:= irregular(2)            [ 2 ];
```

```
    ack_stride_indicator =:= irregular(1)             [ 1 ];
    window_indicator     =:= irregular(1)             [ 1 ];
    ip_id_indicator      =:= irregular(1)             [ 1 ];
    urg_ptr_present      =:= irregular(1)             [ 1 ];
    reserved             =:= compressed_value(1, 0)   [ 1 ];
    ecn_used             =:= one_bit_choice           [ 1 ];
    dscp_present         =:= irregular(1)             [ 1 ];
    ttl_hopl_present     =:= irregular(1)             [ 1 ];
    list_present         =:= irregular(1)             [ 1 ];
    ip_id_behavior_innermost       =:= irregular(2)             [ 2 ];
    urg_flag             =:= irregular(1)             [ 1 ];
    df                   =:= dont_fragment(version.UVALUE)   [ 1 ];
    header_crc           =:= crc7(THIS.UVALUE, THIS.ULENGTH) [ 7 ];
    seq_number           =:=
      variable_length_32_enc(seq_indicator.CVALUE) [ 0, 8, 16, 32 ];
    ack_number           =:=
      variable_length_32_enc(ack_indicator.CVALUE) [ 0, 8, 16, 32 ];
    ack_stride           =:=
      static_or_irreg(ack_stride_indicator.CVALUE, 16)  [ 0, 16 ];
    window               =:=
      static_or_irreg(window_indicator.CVALUE, 16)      [ 0, 16 ];
    ip_id                =:=
      optional_ip_id_lsb(ip_id_behavior_innermost.UVALUE,
                         ip_id_indicator.CVALUE)      [ 0, 8, 16 ];
    urg_ptr              =:=
      static_or_irreg(urg_ptr_present.CVALUE, 16)     [ 0, 16 ];
    dscp                 =:=
      dscp_enc(dscp_present.CVALUE)                   [ 0, 8 ];
    ttl_hopl             =:=
      static_or_irreg(ttl_hopl_present.CVALUE, 8)     [ 0, 8 ];
    options              =:=
      tcp_list_presence_enc(list_present.CVALUE)      [ VARIABLE ];
  }

  // Send LSBs of sequence number
  COMPRESSED rnd_1 {
    discriminator =:= '101110'                        [ 6 ];
    seq_number    =:= lsb(18, 65535)                  [ 18 ];
    msn           =:= lsb(4, 4)                       [ 4 ];
    psh_flag      =:= irregular(1)                    [ 1 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH) [ 3 ];
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_RANDOM) ||
            (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_ZERO));
  }

  // Send scaled sequence number LSBs
  COMPRESSED rnd_2 {
```

```
    discriminator      =:= '1100'                          [ 4 ];
    seq_number_scaled  =:= lsb(4, 7)                       [ 4 ];
    msn                =:= lsb(4, 4)                       [ 4 ];
    psh_flag           =:= irregular(1)                    [ 1 ];
    header_crc         =:= crc3(THIS.UVALUE, THIS.ULENGTH) [ 3 ];
    ENFORCE(payload_size != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_RANDOM) ||
            (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_ZERO));
  }

  // Send acknowledgment number LSBs
  COMPRESSED rnd_3 {
    discriminator =:= '0'                              [ 1 ];
    ack_number    =:= lsb(15, 8191)                    [ 15 ];
    msn           =:= lsb(4, 4)                        [ 4 ];
    psh_flag      =:= irregular(1)                     [ 1 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)  [ 3 ];
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_RANDOM) ||
            (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_ZERO));
  }

  // Send acknowledgment number scaled
  COMPRESSED rnd_4 {
    discriminator      =:= '1101'                          [ 4 ];
    ack_number_scaled  =:= lsb(4, 3)                       [ 4 ];
    msn                =:= lsb(4, 4)                       [ 4 ];
    psh_flag           =:= irregular(1)                    [ 1 ];
    header_crc         =:= crc3(THIS.UVALUE, THIS.ULENGTH) [ 3 ];
    ENFORCE(ack_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_RANDOM) ||
            (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_ZERO));
  }

  // Send ACK and sequence number
  COMPRESSED rnd_5 {
    discriminator =:= '100'                            [ 3 ];
    psh_flag      =:= irregular(1)                     [ 1 ];
    msn           =:= lsb(4, 4)                        [ 4 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)  [ 3 ];
    seq_number    =:= lsb(14, 8191)                    [ 14 ];
    ack_number    =:= lsb(15, 8191)                    [ 15 ];
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_RANDOM) ||
            (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_ZERO));
  }
```

```
  // Send both ACK and scaled sequence number LSBs
  COMPRESSED rnd_6 {
    discriminator      =:= '1010'                          [ 4 ];
    header_crc         =:= crc3(THIS.UVALUE, THIS.ULENGTH) [ 3 ];
    psh_flag           =:= irregular(1)                    [ 1 ];
    ack_number         =:= lsb(16, 16383)                  [ 16 ];
    msn                =:= lsb(4, 4)                       [ 4 ];
    seq_number_scaled  =:= lsb(4, 7)                       [ 4 ];
    ENFORCE(payload_size != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_RANDOM) ||
            (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_ZERO));
  }

  // Send ACK and window
  COMPRESSED rnd_7 {
    discriminator =:= '101111'                         [ 6 ];
    ack_number    =:= lsb(18, 65535)                   [ 18 ];
    window        =:= irregular(16)                    [ 16 ];
    msn           =:= lsb(4, 4)                        [ 4 ];
    psh_flag      =:= irregular(1)                     [ 1 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH) [ 3 ];
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_RANDOM) ||
            (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_ZERO));
  }

  // An extended packet type for seldom-changing fields
  // Can send LSBs of TTL, RSF flags, change ECN behavior, and
  // options list
  COMPRESSED rnd_8 {
    discriminator =:= '10110'                          [ 5 ];
    rsf_flags     =:= rsf_index_enc                    [ 2 ];
    list_present  =:= irregular(1)                     [ 1 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH) [ 7 ];
    msn           =:= lsb(4, 4)                        [ 4 ];
    psh_flag      =:= irregular(1)                     [ 1 ];
    ttl_hopl      =:= lsb(3, 3)                        [ 3 ];
    ecn_used      =:= one_bit_choice                   [ 1 ];
    seq_number    =:= lsb(16, 65535)                   [ 16 ];
    ack_number    =:= lsb(16, 16383)                   [ 16 ];
    options       =:=
      tcp_list_presence_enc(list_present.CVALUE)      [ VARIABLE ];
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_RANDOM) ||
            (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_ZERO));
  }
```

```
  // Send LSBs of sequence number
  COMPRESSED seq_1 {
    discriminator =:= '1010'                                  [ 4 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 4, 3) [ 4 ];
    seq_number    =:= lsb(16, 32767)                          [ 16 ];
    msn           =:= lsb(4, 4)                               [ 4 ];
    psh_flag      =:= irregular(1)                            [ 1 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)         [ 3 ];
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
  }

  // Send scaled sequence number LSBs
  COMPRESSED seq_2 {
    discriminator      =:= '11010'                          [ 5 ];
    ip_id              =:=
      ip_id_lsb(ip_id_behavior_innermost.UVALUE, 7, 3)      [ 7 ];
    seq_number_scaled  =:= lsb(4, 7)                        [ 4 ];
    msn                =:= lsb(4, 4)                        [ 4 ];
    psh_flag           =:= irregular(1)                     [ 1 ];
    header_crc         =:= crc3(THIS.UVALUE, THIS.ULENGTH) [ 3 ];
    ENFORCE(payload_size != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
  }

  // Send acknowledgment number LSBs
  COMPRESSED seq_3 {
    discriminator =:= '1001'                                  [ 4 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 4, 3) [ 4 ];
    ack_number    =:= lsb(16, 16383)                          [ 16 ];
    msn           =:= lsb(4, 4)                               [ 4 ];
    psh_flag      =:= irregular(1)                            [ 1 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)         [ 3 ];
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
  }

  // Send scaled acknowledgment number scaled
  COMPRESSED seq_4 {
    discriminator     =:= '0'                               [ 1 ];
    ack_number_scaled =:= lsb(4, 3)                         [ 4 ];
```

```
    // Due to having very few ip_id bits, no negative offset
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 3, 1) [ 3 ];
    msn               =:= lsb(4, 4)                       [ 4 ];
    psh_flag          =:= irregular(1)                    [ 1 ];
    header_crc        =:= crc3(THIS.UVALUE, THIS.ULENGTH) [ 3 ];
    ENFORCE(ack_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
  }

  // Send ACK and sequence number
  COMPRESSED seq_5 {
    discriminator =:= '1000'                                  [ 4 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 4, 3) [ 4 ];
    ack_number    =:= lsb(16, 16383)                          [ 16 ];
    seq_number    =:= lsb(16, 32767)                          [ 16 ];
    msn           =:= lsb(4, 4)                               [ 4 ];
    psh_flag      =:= irregular(1)                            [ 1 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)         [ 3 ];
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
  }

  // Send both ACK and scaled sequence number LSBs
  COMPRESSED seq_6 {
    discriminator      =:= '11011'                            [ 5 ];
    seq_number_scaled  =:= lsb(4, 7)                          [ 4 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 7, 3) [ 7 ];
    ack_number         =:= lsb(16, 16383)                     [ 16 ];
    msn                =:= lsb(4, 4)                          [ 4 ];
    psh_flag           =:= irregular(1)                       [ 1 ];
    header_crc         =:= crc3(THIS.UVALUE, THIS.ULENGTH)    [ 3 ];
    ENFORCE(payload_size != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
  }

  // Send ACK and window
  COMPRESSED seq_7 {
    discriminator =:= '1100'                                  [ 4 ];
    window        =:= lsb(15, 16383)                          [ 15 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 5, 3) [ 5 ];
```

```
    ack_number    =:= lsb(16, 32767)                          [ 16 ];
    msn           =:= lsb(4, 4)                               [ 4 ];
    psh_flag      =:= irregular(1)                            [ 1 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)         [ 3 ];
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
  }

  // An extended packet type for seldom-changing fields
  // Can send LSBs of TTL, RSF flags, change ECN behavior, and
  // options list
  COMPRESSED seq_8 {
    discriminator =:= '1011'                                  [ 4 ];
    ip_id =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 4, 3) [ 4 ];
    list_present  =:= irregular(1)                            [ 1 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)         [ 7 ];
    msn           =:= lsb(4, 4)                               [ 4 ];
    psh_flag      =:= irregular(1)                            [ 1 ];
    ttl_hopl      =:= lsb(3, 3)                               [ 3 ];
    ecn_used      =:= one_bit_choice                          [ 1 ];
    ack_number    =:= lsb(15, 8191)                           [ 15 ];
    rsf_flags     =:= rsf_index_enc                           [ 2 ];
    seq_number    =:= lsb(14, 8191)                           [ 14 ];
    options       =:=
      tcp_list_presence_enc(list_present.CVALUE)       [ VARIABLE ];
    ENFORCE((ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL) ||
            (ip_id_behavior_innermost.UVALUE ==
             IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
  }
}
```

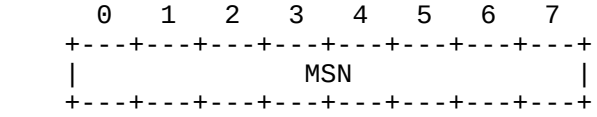
8.3. Feedback Formats and Options

8.3.1. Feedback Formats

This section describes the feedback formats for the ROHC-TCP profile, following the general ROHC feedback format described in Section 5.2.4 of [RFC5795].

All feedback formats carry a field labeled MSN. The MSN field contains LSBs of the MSN control field described in Section 6.1.1. The sequence number to use is the MSN corresponding to the last header that was successfully CRC-8 validated or CRC verified.

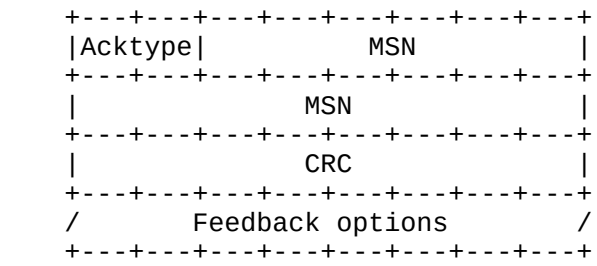
FEEDBACK-1



MSN: The LSB-encoded master sequence number.

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)

MSN: The LSB-encoded master sequence number.

CRC: 8-bit CRC computed over the entire feedback element (as defined in Section 5.3.1.1 of [RFC5795]). For the purpose of computing the CRC, the CRC field is zero. The CRC is calculated using the polynomial defined in [RFC5795].

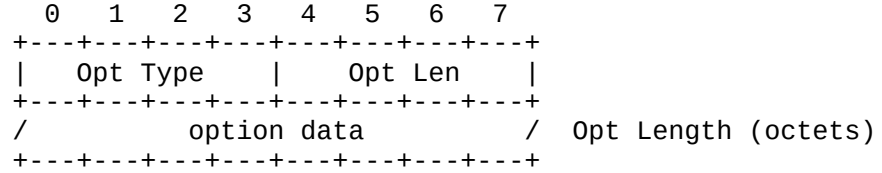
Feedback options: A variable number of feedback options, see Section 8.3.2. Options may appear in any order.

A FEEDBACK-2 of type NACK or STATIC-NACK is always implicitly an acknowledgment for a successfully decompressed packet, which packet corresponds to the MSN of the feedback element, unless the MSN-NOT-VALID option (Section 8.3.2.2) 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 by comparing the result with the CRC carried in the feedback format. If the two are not identical, the feedback element MUST be discarded.

8.3.2. Feedback Options

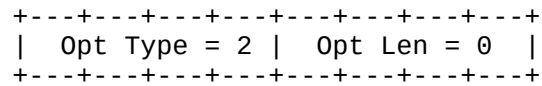
A ROHC-TCP feedback option has variable length and the following general format:



Each ROHC-TCP feedback option can appear at most once within a FEEDBACK-2.

8.3.2.1. The REJECT Option

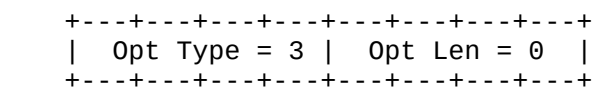
The REJECT option informs the compressor that the decompressor does not have sufficient resources to handle the flow.



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.

8.3.2.2. The MSN-NOT-VALID Option

The MSN-NOT-VALID option indicates that the MSN of the feedback is not valid.

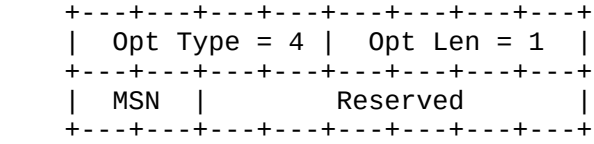


A compressor MUST ignore the MSN of the feedback element when this option is present. Consequently, a NACK or a STATIC-NACK feedback type sent with the MSN-NOT-VALID option is equivalent to a STATIC-NACK with respect to the semantics of the feedback message.

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

8.3.2.3. The MSN Option

The MSN option provides 2 additional bits of MSN.

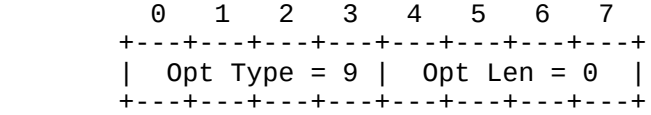


These 2 bits are the least significant bits of the MSN and are thus concatenated with the 14 bits already present in the FEEDBACK-2 format.

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

8.3.2.4. The CONTEXT_MEMORY Feedback Option

The CONTEXT_MEMORY option means 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.

8.3.2.5. Unknown Option Types

If an option type unknown to the compressor is encountered, the compressor MUST continue parsing the rest of the FEEDBACK element, which is possible since the length of the option is explicit, but MUST otherwise ignore the unknown option.

9. Changes from RFC 4996

This RFC revises RFC 4996. It is mostly backwards-compatible with RFC 4996, except for two cases that did not interoperate as described below.

9.1. Functional Changes

- The SACK option compression in [RFC4996] assumed that multiple SACK blocks within the same option would be in sorted order so that the block starts were LSB-encoded from the end of the previous block. This meant that SACK blocks that are not in sorted order could be impossible to compress in some cases. Therefore, the SACK compression in the formal notation has changed and therefore also the bits-on-the-wire.

- The ESP NULL header compression has been deprecated due to interoperability problems with needing to know information from the trailer. The ESP NULL compression was already removed from ROHCv2 [RFC5225] for the same reason and it was considered better to remove it from this profile rather than try to fix the interoperability issue.

9.2. Non-functional Changes

- The way sequential IP-ID compression was described in the FN code was incorrect and the code used for ROHCv2 [RFC5225] has been imported into this specification (e.g., offset is made into a global control field). This does not change the bits-on-the-wire. The only change is how this encoding is described in the formal notation, not how the compression occurs.

- Default encoding for the 'df' and 'ip_id' fields have been added for IPv6 with 0-bit uncompressed format to clarify that these never appear in IPv6.

- The scaled encoding of the Acknowledgment Number and Sequence Number were incorrectly described in the FN code in [RFC4996] and have been updated in the same style as in ROHCv2 [RFC5225]. This does not change the bits-on-the-wire, only the way the compression is described in the FN code.

- The external arguments to ipv4 and co_baseheader have been updated. This is again only a change for FN correctness and does not affect interoperability.

- Errata for [RFC4996] related to minor errors in the FN and textual errors have also been corrected.

10. Security Considerations

A malfunctioning or malicious header compressor could cause the header decompressor to reconstitute packets that do not match the original packets but still have valid IP and TCP headers, and possibly also valid TCP checksums. Such corruption may be detected with end-to-end authentication and integrity mechanisms that will not be affected by the compression. Moreover, this header compression scheme uses an internal checksum for verification of reconstructed headers. This reduces the probability of producing decompressed headers not matching the original ones without this being noticed.

Denial-of-service attacks are possible if an intruder can introduce (for example) bogus IR, CO, 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.

11. IANA Considerations

The reference for the ROHC profile identifier 0x0006 has been updated to reference this document instead of RFC 4996.

A ROHC profile identifier has been reserved by IANA for the profile defined in this document. Profiles 0x0000-0x0005 have previously been reserved; this profile is 0x0006. As for previous ROHC profiles, profile numbers 0xnn06 have been reserved for future updates of this profile.

Profile identifier	Usage	Document
0x0006	ROHC TCP	[RFC6846]
0xnn06	Reserved	

12. Acknowledgments

The authors would like to thank Qian Zhang, Hong Bin Liao, Richard Price, and Fredrik Lindstroem for their work with early versions of this specification. Thanks also to Robert Finking and Carsten Bormann for valuable input and to Carl Knutsson and Gilbert Strom for suggestions and review of the updates made in this document.

Additional thanks: this document was reviewed during working group last-call by committed reviewers Joe Touch and Ted Faber, as well as by Sally Floyd, who provided a review at the request of the Transport Area Directors.

13. References

13.1. Normative References

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

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, 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.

[RFC4164] Pelletier, G., "RObust Header Compression (ROHC): Context Replication for ROHC Profiles", RFC 4164, August 2005.

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

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

[RFC5795] Sandlund, K., Pelletier, G., and L-E. Jonsson, "The RObust Header Compression (ROHC) Framework", RFC 5795, March 2010.

13.2. Informative References

[RFC1144] Jacobson, V., "Compressing TCP/IP headers for low-speed serial links", RFC 1144, February 1990.

[RFC1323] Jacobson, V., Braden, B., and D. Borman, "TCP Extensions for High Performance", RFC 1323, May 1992.

[RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, October 1996.

[RFC2507] Degermark, M., Nordgren, B., and S. Pink, "IP Header Compression", RFC 2507, February 1999.

[RFC2883] Floyd, S., Mahdavi, J., Mathis, M., and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP", RFC 2883, July 2000.

[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.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.

[RFC3759] Jonsson, L-E., "RObust Header Compression (ROHC): Terminology and Channel Mapping Examples", RFC 3759, April 2004.

[RFC4163] Jonsson, L-E., "RObust Header Compression (ROHC): Requirements on TCP/IP Header Compression", RFC 4163, August 2005.

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

[RFC4413] West, M. and S. McCann, "TCP/IP Field Behavior", RFC 4413, March 2006.

[RFC4996] Pelletier, G., Sandlund, K., Jonsson, L-E., and M. West, "RObust Header Compression (ROHC): A Profile for TCP/IP (ROHC-TCP)", RFC 4996, July 2007.

[RFC5225] Pelletier, G. and K. Sandlund, "RObust Header Compression Version 2 (ROHCv2): Profiles for RTP, UDP, IP, ESP and UDP-Lite", RFC 5225, April 2008.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.

Authors' Addresses

Ghyslain Pelletier
InterDigital Communications
1000, Sherbrooke Street West, 10th floor
Montreal, Quebec H3A 3G4
Canada

Phone: +46 (0) 70 609 27 73
EMail: ghyslain.pelletier@interdigital.com

Kristofer Sandlund
Ericsson
Box 920
Lulea SE-971 28
Sweden

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

Lars-Erik Jonsson
Optand 737
Ostersund SE-831 92
Sweden

Phone: +46 70 365 20 58
EMail: lars-erik@lejonsson.com

Mark A West
Siemens/Roke Manor
Roke Manor Research Ltd.
Romsey, Hampshire SO51 0ZN
UK

Phone: +44 1794 833311
EMail: mark.a.west@roke.co.uk
URI: http://www.roke.co.uk