Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 3 August 2026

A. Pelov
IMT Atlantique
Q. Lampin
M. Dumay
Orange

30 January 2026

SCHClet - Modular Use of the SCHC Framework
draft-ietf-schc-schclet-00

Abstract

This document introduces the concept of a SCHClet: a modular sub-function within the SCHC (Static Context Header Compression) framework. Inspired by chiplet architectures in hardware design, a SCHClet encapsulates a specific SCHC function— such as compression, fragmentation, or acknowledgments— as a self-contained unit. This modularization enables tailored implementations that avoid the overhead of deploying a full SCHC stack.

By decomposing SCHC functionality into SCHClets, the framework becomes more adaptable, extensible, and suitable for a wider range of network environments— including, but not limited to, constrained networks. A system using SCHClets remains compliant with the SCHC framework and can interoperate with a full SCHC implementation, provided compatible configuration parameters are used.

Each SCHClet is defined by the SCHC Profiles and configuration parameters necessary for interoperability. It operates within a single Stratum and a single SCHC Instance. For example, a device may implement only the NoAck fragmentation mode as a standalone SCHClet, potentially with fixed parameters. This modular approach simplifies development, reduces resource demands, and provides a framework for future extensibility of the SCHC architecture.

Status of This Memo

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

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

Copyright Notice

Table of Contents

1. Introduction 2
2. Terminology 4
3. Background and Motivation 4
3.1. Simplifications Enabled by SCHClets 5
4. Formal SCHClet definition and validation 5
4.1. SCHC Stratum Header, SCHC Instance, Discriminator 6
5. Use Cases and Examples 6
5.1. IPsec Compression SCHClet 6
5.2. Fragmentation SCHClets 7
5.3. Example: Minimal Fixed-Field Compression SCHClet 7
5.3.1. Functionality 7
5.3.2. Implementation 10
6. Security Considerations 12
7. IANA Considerations 12
8. Normative References 12
Authors' Addresses 12

1. Introduction

The SCHC framework, as defined in RFC8724 and related documents, was originally designed to address the needs of constrained networks by providing efficient header compression and fragmentation. Over time, the addition of new functionalities— such as Compound Acknowledgement and various fragmentation modes— has revealed both the strengths and limitations of a monolithic SCHC implementation.

A SCHClet is a self-contained function of SCHC, which may or may not include all aspects discussed in RFC8724, or other related SCHC RFCs. A SCHClet operates on a single Stratum and on a single SCHC Instance. Notions such as SCHC Stratum Header, SCHC Instance and Discriminator can therefore be omitted in the specification of a SCHClet.

One of the goals for definition of a SCHClet is the fact that parts of SCHC may be applicable to many use-cases, without the need to include the entire SCHC apparatus. This is done to some extent in some of the SCHC RFC technology profiles (e.g. RFC9011), where there is no rule management for example. There is no rule discovery, etc. In that sense, these RFCs use SCHC, built on a specific set of SCHClets.

A full SCHC implementation is supposed to implement all features of RFC8724, and possibly all related RFCs. However, there are more and more RFCs that add supplementary functionalities, such as CompoundAck. In this sense, what is considered a full SCHC implementation today, may not be exhaustive tomorrow. In addition, the SCHC Architecture introduces the notions of SCHC Stratum Header, SCHC Instance and Discriminator, which while useful for a Full SCHC Implementation, could be seen as unnecessary for many of the applications of SCHClets.

A generic SCHC Framework implementation which has all SCHClets implemented MUST be able to interoperate with any SCHClet, provided it has the corresponding configuration. For example, if one IPsec end-point uses a minimal SCHClet implementation, the other end-point may use a full SCHC implementation with the corresponding configuration.

A SCHClet is defined by the SCHC Profiles or configuration parameters that enable interoperability with a Full SCHC Implementation. A SCHClet operates on a single Stratum, within a single SCHC Instance.

For example, the IPsec draft, includes only compression, and with specific compression rules. It may be seen as a SCHClet, using only SCHC Compression from RFC8724, with only a subset of the MOs/CDAs.

This document describes the SCHClet modular approach to the SCHC Framework. Each SCHClet represents an autonomous sub- function of the overall SCHC process. This design enables developers to incorporate only the relevant SCHC functionality for a given application, thereby reducing complexity and resource requirements while retaining the benefits of the SCHC approach. The SCHClet concept provides a way to describe future additions to the SCHC Framework, such as new SCHC Fragmentations, aggregation functions, FEC rule formats.

2. Terminology

* SCHClet: A self-contained unit within the SCHC framework that implements a specific SCHC function or a subset of SCHC operations. A SCHClet may implement aspects defined in RFC8724 or other related SCHC RFCs, and MAY be combined with other SCHClets or integrated into a full SCHC implementation.

* Full SCHC Implementation: An implementation that covers all mandatory aspects of SCHC as defined in RFC8724, potentially extended by additional functionalities introduced in subsequent/ related RFCs.

* Full SCHC Implementation Configuration (Full Configuration): The set of SCHC Profiles/configurations/parameters supported by a Full SCHC Implementation.

* SCHClet Configuration: A subset of a Full Configuration, which are implemented and supported by a given SCHClet. This may be a single SCHC Profile, or a set of such.

3. Background and Motivation

SCHC was developed to provide a robust mechanism for header compression and fragmentation in networks with strict resource constraints. The original design focused on a comprehensive implementation that would cover all aspects of SCHC functionality. However, several trends have emerged:

* Diverse Use Cases: Different applications may require only a subset of SCHC functionalities. For example, some may only need compression while others might focus solely on fragmentation.

* Evolving Standards: As new RFCs extend SCHC functionality (e.g., Compound Acknowledgement, advanced fragmentation techniques), a monolithic implementation risks becoming overly complex.

* Resource Optimization: In constrained environments, it is beneficial to deploy only the necessary SCHC functions to optimize memory, processing power, and energy usage.

The SCHClet concept addresses these challenges by providing a modular approach that decouples individual SCHC functions from a monolithic architecture.

3.1. Simplifications Enabled by SCHClets

By breaking down a SCHC implementation into SCHClets, several simplifications can be realized:

* Exclusion of Rule Management:
  SCHClets can omit complex rule discovery, installation, and update mechanisms. This results in a leaner protocol that focuses on the core function (e.g., compression) without the overhead of managing multiple rule sets.

* Simplification of Rule Management:
  SCHClets can support only a subset of the Rule Management, e.g. read-only access to existing rules.

* Simplification of the use of SCHC Framework: The notions of SCHC Stratum Header, SCHC Instance, and Discriminator MAY be omitted.

* Targeted Functionality: Specific optional functions (such as compound acknowledgments or advanced fragmentation modes) can be encapsulated in separate SCHClets. This approach allows developers to include only the necessary functions for a given use case, simplifying the overall design.

A full SCHC implementation with the right configuration MUST interoperate with a specific SCHClet. The SCHClet in question may not necessarily handle these configurations itself (e.g. it may be a fixed implementation with no parametrization possible on its side), they can be reinterpreted as a full SCHC implementation would. For example, an implementer of a compression SCHClet may never formally use Rule Management, Discriminators, SCHC Header or other notions defined in the SCHC Architecture, these can be inferred by the knowledgeable SCHC practitioner. It is of course RECOMMENDED that a SCHClet provides a complete picture of its use in the context of a full SCHC implementation.

4. Formal SCHClet definition and validation

Formally, a SCHClet is a set of SCHC functions, which when implemented interoperate with a Full SCHC implementation, within a predefined SCHClet Configuration. A SCHClet is therefore fully defined by its corresponding SCHClet Configuration.

Any document specifying a SCHClet MUST include the definition of the supported SCHClet Configuration.

A SCHClet is designed to be interoperable with both minimal and full SCHC implementations. For instance, an endpoint may implement a single SCHClet for a specific function (e.g., IPsec compression) while the corresponding peer uses a comprehensive SCHC implementation that supports multiple SCHClets. Proper configuration and negotiation mechanisms are essential to ensure that both ends correctly interpret the SCHC context.

If a SCHClet is used in a document/specification/implementation, it MAY be cited as: "Using a SCHClet of the SCHC Framework, with the following supported configuration/parameters/profiles: ...".

4.1. SCHC Stratum Header, SCHC Instance, Discriminator

A SCHClet operates on a single Stratum and on a single SCHC Instance. As such, there is no need to specify SCHC Stratum Header, which is always fully elided. In addition, there is no need for a Discriminator. Documents specifying the use of a SCHClet MAY omit the specification of SCHC Stratum Header, SCHC Instance and/or Discriminator considerations.

It is RECOMMENDED if there are more than one SCHC Instances to use a Full SCHC Implementation within the Full SCHC Architecture. While not recommended, a SCHC Stratum Instance MAY be defined as a SCHClet, and combined with other SCHClets to achieve the functionality of a complete SCHC Stratum implementation.

5. Use Cases and Examples

5.1. IPsec Compression SCHClet

An illustrative example is an IPsec draft that leverages SCHC compression:

* Functionality: This SCHClet implements only the compression rules defined in RFC8724 and a limited set of Context-Dependent Attributes (CoDAs).

* Deployment: The implementation is tailored for IPsec environments, focusing solely on compression without engaging in rule management or SCHC Stratum functions.

* Interoperability: Although minimal, this SCHClet must be configured to operate alongside endpoints that might use full SCHC implementations.

As an example, Diet-ESP, as defined in draft-ietf-IPsecme-diet-esp-05 (https://datatracker.ietf.org/doc/html/draft-ietf-IPsecme-diet-esp-05), represents a minimal, streamlined version of the ESP protocol designed for constrained environments. By integrating SCHClets, Diet-ESP can leverage SCHC's compression or fragmentation capabilities in a modular manner, without needing to implement a full SCHC implementation.

5.2. Fragmentation SCHClets

Fragmentation is a core aspect of SCHC, with multiple modes available:

* NoAck Fragmentation SCHClet: Implements a fragmentation scheme without acknowledgement, suitable for low-overhead scenarios.

* Ack-On-Err Fragmentation SCHClet: Incorporates error recovery mechanisms by acknowledging only when errors occur.

* Ack-Always Fragmentation SCHClet: Provides continuous acknowledgement for reliable communication, albeit with increased overhead.

Developers may choose to implement one or more of these SCHClets based on application requirements and network conditions.

5.3. Example: Minimal Fixed-Field Compression SCHClet

The goal of this example is to show how a SCHClet may be a single, constant-time function, of extreme simplicity - but still interoperable with a Full SCHC Implementation.

The example is one that compresses only constant fields in an IPv6 header, under the assumption that specific values are fixed and known a priori at both endpoints. This SCHClet is fully stateless, does not require (nor support) rule management, and can be applied directly at the byte level.

This example is deliberately minimal, to demonstrate how a fully standards-compliant SCHClet can be implemented with only a few lines of code when the deployment assumptions are strongly constrained.

5.3.1. Functionality

This SCHClet targets the first four bytes of the IPv6 header, which correspond to:

* Version (4 bits): 6

* Traffic Class (8 bits): 0

* Flow Label (20 bits): 0

When these fields are known to be constant, they may be fully elided. The SCHClet performs a prefix match against the fixed 4-byte sequence 0x60 00 00 00, and if matched, removes it during compression and restores it during decompression.

This fixed-function approach is applicable in many constrained deployments, particularly when all traffic is known to conform to a common baseline IPv6 configuration.

To simplify the example, we use a RuleID of length 8 bits, with two rules:

* RuleID 01100000 (0x60) - "pass-through", which elides the first byte (which is 0x60), but then adds the ruleID (which is 0x60), effectively making the compression operation a no-operation.

* RuleID 11111111 (0xFF) - "compression", which elides the first 4 bytes

Note that the choice of RuleIDs is wasteful, in the sense that only one bit would be sufficient. This is done to illustrate the simplicity of the implementation, e.g. the code below doesn’t require bit-shifting operations. Also note, that in the code we "change" the buffer with the packet to be compressed, which is done also for the sake of simplicity, but is not a recommended design pattern.

Also note, that this artificial example doesn’t handle the case where a protocol version 1111 with following bits as 1111 is provided to the compressor/decompressor. That may, or may not be a problem, e.g. if it is known that there is only IPv6 traffic processed through this SCHClet, this is not an issue.

Here is the SCHC Context, which enables interoperability between this SCHClet and a Full SCHC Implementation:

```
{
  "schc": {
    "rule": [
      {
        "rule-id-value": 0x60,
        "rule-id-length": 8,
        "nature-compression": {
          "field": [
            {
```

```
                "field-id": "fid-ipv6-version",
                "field-position": 1,
                "direction": "bi",
                "target-value": {
                    "index": 0,
                    "value": 6
                },
                "matching-operator": "mo-equal",
                "comp-decomp-action": "cda-not-sent"
              },
              {
                "field-id": "fid-ipv6-trafficclass",
                "field-position": 1,
                "direction": "bi",
                "target-value": {
                    "index": 0,
                    "value": 0
                },
                "matching-operator": "mo-msb",
                "matching-operator-value":
                {
                    "index": 0,
                    "value": 4
                },
                "comp-decomp-action": "cda-lsb",
                "comp-decomp-action-value":
                {
                    "index": 0,
                    "value": 4
                }
              },
              {
                "field-id": "fid-ipv6-flowlabel",
                "field-position": 1,
                "direction": "bi",
                "target-value": {
                    "index": 0,
                    "value": 0
                },
                "matching-operator": "mo-equal",
                "comp-decomp-action": "cda-value-sent"
              }
            ]
          }
        },
        {
          "rule-id-value": 0xFF,
          "rule-id-length": 8,
```

```
          "nature-compression": {
            "field": [
              {
                "field-id": "fid-ipv6-version",
                "field-position": 1,
                "direction": "bi",
                "target-value": {
                    "index": 0,
                    "value": 6
                },
                "matching-operator": "mo-equal",
                "comp-decomp-action": "cda-not-sent"
              },
              {
                "field-id": "fid-ipv6-trafficclass",
                "field-position": 1,
                "direction": "bi",
                "target-value": {
                    "index": 0,
                    "value": 0
                },
                "matching-operator": "mo-equal",
                "comp-decomp-action": "cda-not-sent"
              },
              {
                "field-id": "fid-ipv6-flowlabel",
                "field-position": 1,
                "direction": "bi",
                "target-value": {
                    "index": 0,
                    "value": 0
                },
                "matching-operator": "mo-equal",
                "comp-decomp-action": "cda-not-sent"
              }
            ]
          }
        }
      ]
    }
  }
```

5.3.2. Implementation

The following C implementation demonstrates this SCHClet in practice:

```
   #include <stdint.h>
   #include <stdio.h>
   #include <string.h>

   #define RULE_ID_PASSTHROUGH 0x60  // 01100000
   #define RULE_ID_COMPRESSED  0xFF  // 11111111

   // Compress packet in-place or pass through, based on content.
   // Returns pointer to compressed packet and sets output length.
   const uint8_t* compress_ipv6(uint8_t *packet, size_t in_len, size_t *out_len) {
       // Match compressible pattern (Version, TC, FL = 0)
       if (in_len >= 4 && packet[0] == 0x60 && packet[1] == 0x00 &&
           packet[2] == 0x00 && packet[3] == 0x00) {

           // We "compress" by replacing the 4th byte with RuleID
           packet[3] = RULE_ID_COMPRESSED;
           *out_len = in_len - 3;
           return &packet[3];
       }

       // Pass-through: return original packet, unchanged
       *out_len = in_len;
       return packet;
   }

   size_t decompress_ipv6(const uint8_t *compressed, size_t comp_len, uint8_t *out_buf, s
ize_t out_max) {
       if (comp_len == 0 || out_max < comp_len + 3) return 0;

       if (compressed[0] == RULE_ID_COMPRESSED) {
           // Insert fixed prefix before compressed payload
           out_buf[0] = 0x60;
           out_buf[1] = 0x00;
           out_buf[2] = 0x00;
           out_buf[3] = 0x00;
           memcpy(out_buf + 4, compressed + 1, comp_len - 1);
           return comp_len - 1 + 4;
       }

       // Rule ID is not compressed — assume passthrough
       memcpy(out_buf, compressed, comp_len);
       return comp_len;
   }

   int main() {
       uint8_t packet[40] = {
           0x60, 0x00, 0x00, 0x00,  // Compressible prefix
           0x3A, 0x40               // Next header, hop limit...
       };
```

```
       size_t in_len = 40;
       size_t comp_len;

       const uint8_t *compressed = compress_ipv6(packet, in_len, &comp_len);
       if (!compressed) {
           printf("Compression failed.\n");
           return 1;
       }

       printf("Compressed length: %zu\n", comp_len);
       printf("Compressed first byte: 0x%02X\n", compressed[0]);

       uint8_t restored[64];
       size_t restored_len = decompress_ipv6(compressed, comp_len, restored, sizeof(resto
red));

       printf("Decompressed first 4 bytes: %02X %02X %02X %02X\n",
              restored[0], restored[1], restored[2], restored[3]);

       return 0;
   }
```

6. Security Considerations

TBD.

7. IANA Considerations

This document does not require any immediate IANA actions.

8. Normative References

[RFC8724] Minaburo, A., Toutain, L., Gomez, C., Barthel, D., and JC. Zuniga, "SCHC: Generic Framework for Static Context Header Compression and Fragmentation", RFC 8724, DOI 10.17487/RFC8724, April 2020, <https://www.rfc-editor.org/info/rfc8724>.

Authors' Addresses

Alexander Pelov
IMT Atlantique
2bis rue de la Chataigneraie
35536 Cesson-Svign
France
Email: alexander.pelov@imt-atlantique.fr

Quentin Lampin
Orange
France
Email: quentin.lampin@orange.com

Marion Dumay
Orange
France
Email: marion.dumay@orange.com