Network Working Group   S. Gössner
Internet-Draft   Fachhochschule Dortmund
Intended status: Standards Track   C. Bormann, Ed.
Expires: 13 January 2021   Universität Bremen TZI
                                                             12 July 2020

JSONPath -- XPath for JSON
draft-goessner-dispatch-jsonpath-00

Abstract

insert abstract here

Contributing

This document picks up the popular JSONPath specification dated 2007-02-21 and provides a more normative definition for it. It is intended as a submission to the IETF DISPATCH WG, in order to find the right way to complete standardization of this specification. In its current state, it is a strawman document showing what needs to be covered.

Comments and issues can be directed at the github repository _insert repo here_ as well as (for the time when the more permanent home is being decided) at the dispatch@ietf.org mailing list.

Status of This Memo

This Internet-Draft will expire on 13 January 2021.

Copyright Notice

Table of Contents

1. Introduction

This document picks up the popular JSONPath specification dated 2007-02-21 [JSONPath-orig] and provides a more normative definition for it. It is intended as a submission to the IETF DISPATCH WG, in order to find the right way to complete standardization of this specification. In its current state, it is a strawman document showing what needs to be covered.

1.1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The grammatical rules in this document are to be interpreted as described in [RFC5234].

The terminology of [RFC8259] applies.

Data Item: A structure complying to the generic data model of JSON, i.e., composed of containers such as arrays and maps (JSON objects), and of atomic data such as null, true, false, numbers, and text strings.

Object: Used in its generic sense, e.g., for programming language objects. When a JSON Object as defined in [RFC8259] is meant, we specifically say JSON Object.

Query: Short name for JSONPath expression.

Argument: Short name for the JSON data item a JSONPath expression is applied to.

Output Path: A simple form of JSONPath expression that identifies a Position by providing a query that results in exactly that position. Similar to, but syntactically different from, a JSON Pointer [RFC6901].

Position: A JSON data item identical to or nested within the JSON data item to which the query is applied to, expressed either by the value of that data item or by providing a JSONPath Output Path.

1.2. Inspired by XPath

A frequently emphasized advantage of XML is the availability of plenty tools to analyse, transform and selectively extract data out of XML documents. [XPath] is one of these powerful tools.

In 2007, the need for something solving the same class of problems for the emerging JSON community became apparent, specifically for:

- Finding data interactively and extracting them out of [RFC8259] data items without special scripting.

* Specifying the relevant parts of the JSON data in a request by a client, so the server can reduce the data in the server response, minimizing bandwidth usage.

So how does such a tool look like when done for JSON? When defining a JSONPath, how should expressions look like?

The XPath expression

```
/store/book[1]/title
```

looks like

```
x.store.book[0].title
```

or

```
x['store']['book'][0]['title']
```

in popular programming languages such as JavaScript, Python and PHP, with a variable x holding the JSON data item. Here we observe that such languages already have a fundamentally XPath-like feature built in.

The JSONPath tool in question should:

* be naturally based on those language characteristics.

* cover only essential parts of XPath 1.0.

* be lightweight in code size and memory consumption.

* be runtime efficient.

1.3. Overview of JSONPath Expressions

JSONPath expressions always apply to a JSON data item in the same way as XPath expressions are used in combination with an XML document. Since a JSON data item is usually anonymous and doesn't necessarily have a "root member object", JSONPath used the abstract name "$" to refer to the top level object of the data item.

JSONPath expressions can use the _dot-notation_

```
$.store.book[0].title
```

or the _bracket-notation_

```
$['store']['book'][0]['title']
```

for paths input to a JSONPath processor. Where a JSONPath processor uses JSONPath expressions for internal purposes or as output paths, these will always be converted to the more general _bracket-notation_.

JSONPath allows the wildcard symbol "*" for member names and array indices. It borrows the descendant operator ".." from [E4X] and the array slice syntax proposal "[start:end:step]" [SLICE] from ECMASCRIPT 4.

JSONPath can employ an _underlying scripting language_, expressions of which, written in parentheses: "(<expr>)", can be used as an alternative to explicit names or indices as in:

```
$.store.book[(@.length-1)].title
```

The symbol "@" is used for the current object. Filter expressions are supported via the syntax "?(<boolean expr>)" as in

```
$.store.book[?(@.price < 10)].title
```

Here is a complete overview and a side by side comparison of the JSONPath syntax elements with its XPath counterparts.

XPath	JSONPath	Description
/	$	the root object/element
.	@	the current object/element
/	"." or "[]"	child operator
..	n/a	parent operator
//	..	nested descendants. JSONPath borrows this syntax from E4X.
*	*	wildcard. All objects/elements regardless of their names.
@	n/a	attribute access. JSON data items don't have attributes.
[]	[]	subscript operator. XPath uses it to iterate over element collections and for predicates. In JavaScript and JSON it is the native array operator.
|	[,]	Union operator in XPath results in a combination of node sets. JSONPath allows alternate names or array indices as a set.
n/a	[start:end:step]	array slice operator borrowed from ES4.
[]	?()	applies a filter (script) expression.
n/a	()	script expression, using the underlying script engine.
()	n/a	grouping in Xpath

Table 1: Overview over JSONPath, comparing to XPath

XPath has a lot more to offer (location paths in unabbreviated syntax, operators and functions) than listed here. Moreover there is a significant difference how the subscript operator works in Xpath and JSONPath:

- Square brackets in XPath expressions always operate on the _node set_ resulting from the previous path fragment. Indices always start at 1.

- With JSONPath, square brackets operate on the _object_ or _array_ addressed by the previous path fragment. Indices always start at 0.

2. JSONPath Examples

This section provides some more examples for JSONPath expressions. The examples are based on a simple JSON data item patterned after a typical XML example representing a bookstore (that also has bicycles):

```
{ "store": {
    "book": [
      { "category": "reference",
        "author": "Nigel Rees",
        "title": "Sayings of the Century",
        "price": 8.95
      },
      { "category": "fiction",
        "author": "Evelyn Waugh",
        "title": "Sword of Honour",
        "price": 12.99
      },
      { "category": "fiction",
        "author": "Herman Melville",
        "title": "Moby Dick",
        "isbn": "0-553-21311-3",
        "price": 8.99
      },
      { "category": "fiction",
        "author": "J. R. R. Tolkien",
        "title": "The Lord of the Rings",
        "isbn": "0-395-19395-8",
        "price": 22.99
      }
    ],
    "bicycle": {
      "color": "red",
      "price": 19.95
    }
  }
}
```

Figure 1: Example JSON data item

The examples in Table 2 presume an underlying script language that allows obtaining the number of items in an array, testing for the presence of a map member, and performing numeric comparisons of map member values with a constant.

XPath	JSONPath	Result
/store/book/author	$.store.book[*].author	the authors of all books in the store
//author	$..author	all authors
/store/*	$.store.*	all things in store, which are some books and a red bicycle.
/store//price	$.store..price	the price of everything in the store.
//book[3]	$..book[2]	the third book
//book[last()]	"$..book[(@.length-1)]" "$..book[-1:]"	the last book in order.
//book[position()<3]	"$..book[0,1]" "$..book[:2]"	the first two books
//book[isbn]	$..book[?(@.isbn)]	filter all books with isbn number
//book[price<10]	$..book[?(@.price<10)]	filter all books cheapier than 10
//*	$..*	all Elements in XML document. All members of JSON data item.

Table 2: Example JSONPath expressions applied to the example JSON data item

3. Detailed definition

[TBD: This section needs to be fleshed out in detail. The text given here is intended to give the flavor of that detail, not to be the actual definition that is to be defined.]

JSONPath expressions, "queries" for short in this specification, are character strings, represented in UTF-8 unless otherwise required by the context in which they are used.

When applied to a JSON data item, a query returns a (possibly empty) list of "positions" in the data item that match the JSONPath expression.

```
JSONPath = root *(step)
root = "$"

step = ".." name ; nested descendants
     / "." name ; child (dot notation)
     / "[" value-expression *("," value-expression) "]"
        ; child[ren] (bracket notation)
     / "[" value-expression *2(":" value-expression) "]"  ; (slice)
value-expression = *DIGIT / name
                 / script-expression / filter-expression
name = "'" text "'"
     / "*" ; wildcard
script-expression = "(" script ")"
filter-expression = "?(" script ")"
script = <To be defined in the course of standardization>
text = <any text, restrictions to be defined>
DIGIT = %x30-39
```

Figure 2: ABNF definition for JSONPath

Within a script, @ stands for the position under consideration. [TBD: define underlying scripting language, if there is to be a standard one]

[TBD: define handling of spaces]

A JSONPath starts at the root of the argument; the "current list" of positions is initialized to that root. Each step applies the semantics of that step to each of the positions in the "current list", returning another list; the "current list" is replaced by the concatenation of all these returned lists, and the next step begins. When all steps have been performed, the "current list" is returned, depending on the choices of the context either as a list of data items or as a list of output paths. [TBD: define the order of that list]

[TBD: Define all the steps]

[TBD: Define details of Output Path]

4. Discussion

* Currently only single quotes allowed inside of JSONPath expressions.

* Script expressions inside of JSONPath locations are currently not recursively evaluated by jsonPath. Only the global "$" and local "@" symbols are expanded by a simple regular expression.

* An alternative for jsonPath to return false in case of no match may be to return an empty array in future. [This is already done in the above.]

5. IANA considerations

TBD: Define a media type for JSON Path expressions.

6. References

6.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

6.2. Informative References

[E4X] ISO, "Information technology -- ECMAScript for XML (E4X) specification", ISO/IEC 22537:2006 , 2006.

[JSON-PHP] "JSON-PHP", January 2005, <http://mike.teczno.com/json.html>.

[JSONPath-impl] "jsonpath (Downloads)", n.d., <https://code.google.com/archive/p/jsonpath/downloads>.

[JSONPath-orig] Gössner, S., "JSONPath - XPath for JSON", 21 February 2007, <https://goessner.net/articles/JsonPath/>.

[RFC6901] Bryan, P., Ed., Zyp, K., and M. Nottingham, Ed., "JavaScript Object Notation (JSON) Pointer", RFC 6901, DOI 10.17487/RFC6901, April 2013, <https://www.rfc-editor.org/info/rfc6901>.

[SLICE] "Slice notation", n.d., <https://github.com/tc39/proposal-slice-notation>.

[XPath] Berglund, A., Boag, S., Chamberlin, D., Fernandez, M., Kay, M., Robie, J., and J. Simeon, "XML Path Language (XPath) 2.0 (Second Edition)", World Wide Web Consortium Recommendation REC-xpath20-20101214, 14 December 2010, <http://www.w3.org/TR/2010/REC-xpath20-20101214>.

Appendix A. Early JSONPath implementations

This appendix has been copied from the similar section in [JSONPath-orig], with few changes. It is informative, intended to supply more examples and give an impression for what could be a typical JSONPath API.

A.1. Implementation

JSONPath is implemented in JavaScript for client-side usage and ported over to PHP for use on the server.

A.2. Usage

All you need to do is downloading either of the files

- "jsonpath.js" [JSONPath-impl]
- "jsonpath.php" [JSONPath-impl]

include it in your program and use the simple API consisting of one single function.

```
jsonPath(obj, expr [, args])
```

A.3. Parameters

obj (object|array): Object representing the JSON data item.

expr (string): JSONPath expression string.

args (object|undefined): Object controlling path evaluation and output. Currently only one member is supported.

args.resultType ("VALUE"|"PATH"): causes the result to be either matching values (default) or normalized path expressions.

A.4. Return value

(array|false): Array holding either values or normalized path expressions matching the input path expression, which can be used for lazy evaluation. "false" in case of no match.

A.5. JavaScript Example

```
var o = { /*...*/ },  // the 'store' JSON object from above
    res1 = jsonPath(o, "$..author").toJSONString(),
    res2 = jsonPath(o, "$..author",
                    {resultType:"PATH"}).toJSONString();
```

A.6. PHP example

We need here to convert the JSON string to a PHP array first. I am using Michal Migurski's JSON parser [JSON-PHP] for that.

```
require_once('json.php');      // JSON parser
require_once('jsonpath.php');  // JSONPath evaluator

$json = '{ ... }';  // JSON data item from above

$parser = new Services_JSON(SERVICES_JSON_LOOSE_TYPE);
$o = $parser->decode($json);
$match1 = jsonPath($o, "$..author");
$match2 = jsonPath($o, "$..author",
                   array("resultType" => "PATH"));
$res1 = $parser->encode($match1);
$res2 = $parser->encode($match2);
```

A.7. Results

Both JavaScript and PHP example result in the following JSON arrays (as strings):

```
res1:
[ "Nigel Rees",
  "Evelyn Waugh",
  "Herman Melville",
  "J. R. R. Tolkien"
]
res2:
[ "$['store']['book'][0]['author']",
  "$['store']['book'][1]['author']",
  "$['store']['book'][2]['author']",
  "$['store']['book'][3]['author']"
]
```

Please note that the return value of jsonPath is an array, which is also a valid JSON data item. So you might want to apply jsonPath to the resulting data item again or use one of your favorite array methods as sort with it.

Acknowledgements

This specification is based on Stefan Gössner's original online article defining JSONPath [JSONPath-orig].

The books example was taken from http://coli.lili.uni-bielefeld.de/~andreas/Seminare/sommer02/books.xml -- a dead link now.

Authors' Addresses

Stefan Gössner
Fachhochschule Dortmund
Sonnenstraße 96
D-44139 Dortmund
Germany

Email: stefan.goessner@fh-dortmund.de

Carsten Bormann (editor)
Universität Bremen TZI
Postfach 330440
D-28359 Bremen
Germany

Phone: +49-421-218-63921
Email: cabo@tzi.org

DISPATCH
Internet-Draft
Updates: 3405 (if approved)
Intended status: Best Current Practice
Expires: 18 April 2021

T. Hardie
15 October 2020

Updated registration rules for URI.ARPA
draft-hardie-dispatch-rfc3405-update-04

Abstract

This document updates RFC 3405 by removing references to the IETF tree from the procedures for requesting that a URI scheme be inserted into the uri.arpa zone.

Status of This Memo

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

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

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

This Internet-Draft will expire on 18 April 2021.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction

Part five of the Dynamic Delegation Discovery System (DDDS), RFC 3405 [RFC3405], describes the registration procedures for assignments in URI.ARPA. The document requires that registrations be in the "IETF tree" of URI registrations. The use of URI scheme name trees was defined in RFC 2717 [RFC2717] but discontinued by RFC 4395 [RFC4395] and its successors. Since the use of trees was discontinued, there is no way in the current process set out in BCP 35 [RFC7595] to meet the requirement to register within that tree.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119][RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Updated Requirements

This document removes the normative requirement from RFC 3405 for registrations in URI.ARPA to be from the IETF URI Tree.

All registrations in URI.ARPA MUST now be for schemes which are permanent registrations, as they are described in BCP 35.

3. IANA Considerations

This entire document is updated instructions to IANA.

4. Security Considerations

This update does not change the Security Considerations in RFC 3405

5. References

5.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3405] Mealling, M., "Dynamic Delegation Discovery System (DDDS) Part Five: URI.ARPA Assignment Procedures", BCP 65, RFC 3405, DOI 10.17487/RFC3405, October 2002, <https://www.rfc-editor.org/info/rfc3405>.

[RFC7595] Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, DOI 10.17487/RFC7595, June 2015, <https://www.rfc-editor.org/info/rfc7595>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

5.2. Informative References

[RFC2717] Petke, R. and I. King, "Registration Procedures for URL Scheme Names", RFC 2717, DOI 10.17487/RFC2717, November 1999, <https://www.rfc-editor.org/info/rfc2717>.

[RFC4395] Hansen, T., Hardie, T., and L. Masinter, "Guidelines and Registration Procedures for New URI Schemes", RFC 4395, DOI 10.17487/RFC4395, February 2006, <https://www.rfc-editor.org/info/rfc4395>.

Author’s Address

Ted Hardie

Email: ted.ietf@gmail.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: May 20, 2021

E. Omara
J. Uberti
Google
A. GOUAILLARD
S. Murillo
CoSMo Software
November 16, 2020

Secure Frame (SFrame)
draft-omara-sframe-01

Abstract

This document describes the Secure Frame (SFrame) end-to-end encryption and authentication mechanism for media frames in a multiparty conference call, in which central media servers (SFUs) can access the media metadata needed to make forwarding decisions without having access to the actual media. The proposed mechanism differs from other approaches through its use of media frames as the encryptable unit, instead of individual RTP packets, which makes it more bandwidth efficient and also allows use with non-RTP transports.

Status of This Memo

This Internet-Draft will expire on May 20, 2021.

Copyright Notice

Table of Contents

1. Introduction

Modern multi-party video call systems use Selective Forwarding Unit (SFU) servers to efficiently route RTP streams to call endpoints based on factors such as available bandwidth, desired video size, codec support, and other factors. In order for the SFU to work properly though, it needs to be able to access RTP metadata and RTCP feedback messages, which is not possible if all RTP/RTCP traffic is end-to-end encrypted.

As such, two layers of encryptions and authentication are required:

1. Hop-by-hop (HBH) encryption of media, metadata, and feedback messages between the the endpoints and SFU

2. End-to-end (E2E) encryption of media between the endpoints

While DTLS-SRTP can be used as an efficient HBH mechanism, it is inherently point-to-point and therefore not suitable for a SFU context. In addition, given the various scenarios in which video calling occurs, minimizing the bandwidth overhead of end-to-end encryption is also an important goal.

This document proposes a new end-to-end encryption mechanism known as SFrame, specifically designed to work in group conference calls with SFUs.

```
     +-------------------------------+--------------------------------+^+
     |V=2|P|X|  CC   |M|     PT      |       sequence number          | |
     +-------------------------------+--------------------------------+ |
     |                           timestamp                            | |
     +----------------------------------------------------------------+ |
     |           synchronization source (SSRC) identifier             | |
     |=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=| |
     |            contributing source (CSRC) identifiers              | |
     |                               ....                             | |
     +----------------------------------------------------------------+ |
     |                   RTP extension(s) (OPTIONAL)                  | |
   +^---------------------+-------------------------------------------+ |
   | |   payload header   |                                           | |
   | +--------------------+     payload  ...                          | |
   | |                                                                | |
   +^+----------------------------------------------------------------+^+
   | :                       authentication tag                       : |
   | +----------------------------------------------------------------+ |
   |                                                                    |
   ++ Encrypted Portion                       Authenticated Portion +--+
```

SRTP packet format

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

SFU:	Selective Forwarding Unit (AKA RTP Switch)
IV:	Initialization Vector
MAC:	Message Authentication Code
E2EE:	End to End Encryption
HBH:	Hop By Hop
KMS:	Key Management System

3. Goals

SFrame is designed to be a suitable E2EE protection scheme for conference call media in a broad range of scenarios, as outlined by the following goals:

1. Provide an secure E2EE mechanism for audio and video in conference calls that can be used with arbitrary SFU servers.
2. Decouple media encryption from key management to allow SFrame to be used with an arbitrary KMS.
3. Minimize packet expansion to allow successful conferencing in as many network conditions as possible.
4. Independence from the underlying transport, including use in non-RTP transports, e.g., WebTransport.
5. When used with RTP and its associated error resilience mechanisms, i.e., RTX and FEC, require no special handling for RTX and FEC packets.
6. Minimize the changes needed in SFU servers.
7. Minimize the changes needed in endpoints.
8. Work with the most popular audio and video codecs used in conferencing scenarios.

4. SFrame

We propose a frame level encryption mechanism that provides effective end-to-end encryption, is simple to implement, has no dependencies on RTP, and minimizes encryption bandwidth overhead. Because SFrame encrypts the full frame, rather than individual packets, bandwidth overhead is reduced by having a single IV and authentication tag for each media frame.

Also, because media is encrypted prior to packetization, the encrypted frame is packetized using a generic RTP packetizer instead of codec-dependent packetization mechanisms. With this move to a generic packetizer, media metadata is moved from codec-specific mechanisms to a generic frame RTP header extension which, while visible to the SFU, is authenticated end-to-end. This extension includes metadata needed for SFU routing such as resolution, frame beginning and end markers, etc.

The generic packetizer splits the E2E encrypted media frame into one or more RTP packets and adds the SFrame header to the beginning of the first packet and an auth tag to the end of the last packet.

```
      +--------------------------------------------------------+
      |                                                        |
      |  +----------+      +------------+      +------------+  |
      |  |          |      |   SFrame   |      |Packetizer  |  |      DTLS+SRTP
      |  | Encoder  +----->+    Enc     +----->+            +---------------------
----+
 ,+.  |  |          |      |            |      |            |  |   +--+  +--+  +--
+   |
 `|'  |  +----------+      +-----+------+      +------------+  |   |  |  |  |  |
|   |
 /|\  |                          ^                             |   |  |  |  |  |
|   |
  +   |                          |                             |   |  |  |  |  |
|   |
 / \  |                          |                             |   +--+  +--+  +--
+   |
Alice |                    +-----+------+                      |  Encrypted Packe
ts  |
      |                    |Key Manager |                      |
    |
      |                    +------------+                      |
    |
      |                         ||                             |
    |
      |                         ||                             |
    |
      |                         ||                             |
    |
      +--------------------------------------------------------+
    |
                                ||
    |
                                ||
    v
                           +------------+                                       +-
----+------+
            E2EE channel   |  Messaging |                                       |
  Media    |
              via the      |  Server    |                                       |
  Server   |
          Messaging Server |            |                                       |
           |
                           +------------+                                       +-
----+------+
                                ||
    |
                                ||
    |
      +--------------------------------------------------------+
    |
      |                         ||                             |
    |
      |                         ||                             |
    |
      |                         ||                             |
    |
      |                    +------------+                      |
    |
```

```
      |                    |Key Manager |                      |
    |
 ,+.  |                    +-----+------+                      |   Encrypted Packe
ts  |
 `|'  |                          |                             |   +--+  +--+  +--
+   |
 /|\  |                          |                             |   |  |  |  |  |
|   |
  +   |                          v                             |   |  |  |  |  |
|   |
 / \  |  +----------+      +-----+------+      +------------+  |   |  |  |  |  |
|   |
 Bob  |  |          |      |   SFrame   |      |   De+      |  |   +--+  +--+  +--
+   |
      |  | Decoder  +<-----+    Dec     +<-----+Packetizer  +<--------------------
----+
      |  |          |      |            |      |            |  |       DTLS+SRTP
      |  +----------+      +------------+      +------------+  |
      |                                                        |
      +--------------------------------------------------------+
```

The E2EE keys used to encrypt the frame are exchanged out of band using a secure E2EE channel.

4.1. SFrame Format

```
     +------------+------------------------------------------+^+
     |S|LEN|X|KID |         Frame Counter                    | |
   +^+------------+------------------------------------------+ |
   | |                                                       | |
   | |                                                       | |
   | |                                                       | |
   | |                                                       | |
   | |                  Encrypted Frame                      | |
   | |                                                       | |
   | |                                                       | |
   | |                                                       | |
   | |                                                       | |
   +^+-------------------------------------------------------+^+
   | |                 Authentication Tag                    | |
   | +-------------------------------------------------------+ |
   |                                                           |
   |                                                           |
   +----+Encrypted Portion            Authenticated Portion+---+
```

4.2. SFrame Header

Since each endpoint can send multiple media layers, each frame will have a unique frame counter that will be used to derive the encryption IV. The frame counter must be unique and monotonically increasing to avoid IV reuse.

As each sender will use their own key for encryption, so the SFrame header will include the key id to allow the receiver to identify the key that needs to be used for decrypting.

Both the frame counter and the key id are encoded in a variable length format to decrease the overhead, so the first byte in the Sframe header is fixed and contains the header metadata with the following format:

```
    0 1 2 3 4 5 6 7
   +-+-+-+-+-+-+-+-+
   |S|LEN  |X|  K  |
   +-+-+-+-+-+-+-+-+
```

SFrame header metadata

Signature flag (S): 1 bit This field indicates the payload contains a signature if set. Counter Length (LEN): 3 bits This field indicates the length of the CTR fields in bytes. Extended Key Id Flag (X): 1 bit Indicates if the key field contains the key id or the key length.

Key or Key Length: 3 bits This field contains the key id (KID) if the X flag is set to 0, or the key length (KLEN) if set to 1.

If X flag is 0 then the KID is in the range of 0-7 and the frame counter (CTR) is found in the next LEN bytes:

```
     0 1 2 3 4 5 6 7
    +-+-+-+-+-+-+-+-+---------------------------------+
    |S|LEN  |0| KID |    CTR... (length=LEN)          |
    +-+-+-+-+-+-+-+-+---------------------------------+
```

Key id (KID): 3 bits The key id (0-7). Frame counter (CTR): (Variable length) Frame counter value up to 8 bytes long.

if X flag is 1 then KLEN is the length of the key (KID), that is found after the SFrame header metadata byte. After the key id (KID), the frame counter (CTR) will be found in the next LEN bytes:

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+---------------------------+---------------------------+
|S|LEN  |1|KLEN |   KID... (length=KLEN)    |    CTR... (length=LEN)    |
+-+-+-+-+-+-+-+-+---------------------------+---------------------------+
```

Key length (KLEN): 3 bits The key length in bytes. Key id (KID): (Variable length) The key id value up to 8 bytes long. Frame counter (CTR): (Variable length) Frame counter value up to 8 bytes long.

4.3. Encryption Schema

SFrame encryption uses an AEAD encryption algorithm and hash function defined by the ciphersuite in use (see Section 4.5). We will refer to the following aspects of the AEAD algorithm below:

- "AEAD.Encrypt" and "AEAD.Decrypt" - The encryption and decryption functions for the AEAD. We follow the convention of RFC 5116 [RFC5116] and consider the authentication tag part of the ciphertext produced by "AEAD.Encrypt" (as opposed to a separate field as in SRTP [RFC3711]).

- "AEAD.Nk" - The size of a key for the encryption algorithm, in bytes

- "AEAD.Nn" - The size of a nonce for the encryption algorithm, in bytes

4.3.1. Key Selection

Each SFrame encryption or decryption operation is premised on a single secret "base_key", which is labeled with an integer KID value signaled in the SFrame header.

The sender and receivers need to agree on which key should be used for a given KID. The process for provisioning keys and their KID values is beyond the scope of this specification, but its security properties will bound the assurances that SFrame provides. For example, if SFrame is used to provide E2E security against intermediary media nodes, then SFrame keys MUST be negotiated in a way that does not make them accessible to these intermediaries.

For each known KID value, the client stores the corresponding symmetric key "base_key". For keys that can be used for encryption, the client also stores the next counter value CTR to be used when encrypting (initially 0).

When encrypting a frame, the application specifies which KID is to be used, and the counter is incremented after successful encryption. When decrypting, the "base_key" for decryption is selected from the available keys using the KID value in the SFrame Header.

A given key MUST NOT be used for encryption by multiple senders. Such reuse would result in multiple encrypted frames being generated with the same (key, nonce) pair, which harms the protections provided by many AEAD algorithms. Implementations SHOULD mark each key as usable for encryption or decryption, never both.

Note that the set of available keys might change over the lifetime of a real-time session. In such cases, the client will need to manage key usage to avoid media loss due to a key being used to encrypt before all receivers are able to use it to decrypt. For example, an application may make decryption-only keys available immediately, but delay the use of encryption-only keys until (a) all receivers have acknowledged receipt of the new key or (b) a timeout expires.

4.3.2. Key Derivation

SFrame encrytion and decryption use a key and salt derived from the "base_key" associated to a KID. Given a "base_key" value, the key and salt are derived using HKDF [RFC5869] as follows:

```
sframe_secret = HKDF-Extract(K, ’SFrame10’)
sframe_key = HKDF-Expand(sframe_secret, ’key’, AEAD.Nk)
sframe_salt = HKDF-Expand(sframe_secret, ’salt’, AEAD.Nn)
```

The hash function used for HKDF is determined by the ciphersuite in use.

4.3.3. Encryption

After encoding the frame and before packetizing it, the necessary media metadata will be moved out of the encoded frame buffer, to be used later in the RTP generic frame header extension. The encoded frame, the metadata buffer and the frame counter are passed to SFrame encryptor.

SFrame encryption uses the AEAD encryption algorithm for the ciphersuite in use. The key for the encryption is the "sframe_key" and the nonce is formed by XORing the "sframe_salt" with the current counter, encoded as a big-endian integer of length "AEAD.Nn".

The encryptor forms an SFrame header using the S, CTR, and KID values provided. The encoded header is provided as AAD to the AEAD encryption operation, with any frame metadata appended.

```
def encrypt(S, CTR, KID, frame_metadata, frame):
  sframe_key, sframe_salt = key_store[KID]

  frame_ctr = encode_big_endian(CTR, AEAD.Nn)
  frame_nonce = xor(sframe_salt, frame_ctr)

  header = encode_sframe_header(S, CTR, KID)
  frame_aad = header + frame_metadata

  encrypted_frame = AEAD.Encrypt(sframe_key, frame_nonce, frame_aad, frame)
  return header + encrypted_frame
```

The encrypted payload is then passed to a generic RTP packetized to construct the RTP packets and encrypt it using SRTP keys for the HBH encryption to the media server.

```
      +----------------+  +---------------+
      | frame metadata |  |               |
      +-------+--------+  |               |
              |           |     frame     |
              |           |               |
              |           |               |
              |           +-------+-------+
              |                   |
   header ----+------------------>| AAD
   +-----+                        |
   |  S  |                        |
   +-----+                        |
   | KID +--+--> sframe_key ----->| Key
   |     |  |                     |
   |     |  +--> sframe_salt -+   |
   +-----+                    |   |
   | CTR +--------------------+-->| Nonce
   |     |                        |
   |     |                        |
   +-----+                        |
      |                       AEAD.Encrypt
      |                           |
      |                           V
      |                   +-------+-------+
      |                   |               |
      |                   |               |
      |                   |   encrypted   |
      |                   |     frame     |
      |                   |               |
      |                   |               |
      |                   +-------+-------+
      |                           |
      |                  generic RTP packetize
      |                           |
      |                           v
      V
   +---------------+      +---------------+     +---------------+
   | SFrame header |      |               |     |               |
   +---------------+      |               |     |               |
   |               |      |  payload 2/N  |     |  payload N/N  |
   |  payload 1/N  |      |               |     |               |
   |               |      |               |     |               |
   +---------------+      +---------------+     +---------------+
```

Encryption flow

4.3.4. Decryption

The receiving clients buffer all packets that belongs to the same frame using the frame beginning and ending marks in the generic RTP frame header extension, and once all packets are available, it passes it to SFrame for decryption. The KID field in the SFrame header is used to find the right key for the encrypted frame.

```
def decrypt(frame_metadata, sframe):
  header, encrypted_frame = split_header(sframe)
  S, CTR, KID = parse_header(header)

  sframe_key, sframe_salt = key_store[KID]

  frame_ctr = encode_big_endian(CTR, AEAD.Nn)
  frame_nonce = xor(sframe_salt, frame_ctr)
  frame_aad = header + frame_metadata

  return AEAD.Decrypt(sframe_key, frame_nonce, frame_aad, encrypted_frame)
```

For frames that are failed to decrypt because there is key available for the KID in the SFrame header, the client MAY buffer the frame and retry decryption once a key with that KID is received.

4.3.5. Duplicate Frames

Unlike messaging application, in video calls, receiving a duplicate frame doesn’t necessary mean the client is under a replay attack, there are other reasons that might cause this, for example the sender might just be sending them in case of packet loss. SFrame decryptors use the highest received frame counter to protect against this. It allows only older frame pithing a short interval to support out of order delivery.

4.4. Authentication

Every client in the call knows the secret key for all other clients so it can decrypt their traffic, it also means a malicious client can impersonate any other client in the call by using the victim key to encrypt their traffic. This might not be a problem for consumer application where the number of clients in the call is small and users know each others, however for enterprise use case where large conference calls are common, an authentication mechanism is needed to protect against malicious users. This authentication will come with extra cost.

Adding a digital signature to each encrypted frame will be an overkill, instead we propose adding signature over multiple frames.

The signature is calculated by concatenating the authentication tags of the frames that the sender wants to authenticate (in reverse sent order) and signing it with the signature key. Signature keys are exchanged out of band along the encryption keys.

```
Signature = Sign(Key, AuthTag(Frame N) || AuthTag(Frame N-1) || ...|| AuthTag(Fra
me N-M))
```

The authentication tags for the previous frames covered by the signature and the signature itself will be appended at end of the frame, after the current frame authentication tag, in the same order that the signature was calculated, and the SFrame header metadata signature bit (S) will be set to 1.

```
       +^ +------------------+
       |  | SFrame header S=1|
       |  +------------------+
       |  |  Encrypted       |
       |  |  payload         |
       |  |                  |
       |^ +------------------+ ^+
       |  |  Auth Tag N      |  |
       |  +------------------+  |
       |  |  Auth Tag N-1    |  |
       |  +------------------+  |
       |  |    ........      |  |
       |  +------------------+  |
       |  |  Auth Tag N-M    |  |
       |  +------------------+ ^|
       |  | NUM | Signature  :  |
       |  +-----+            +  |
       |  :                  |  |
       |  +------------------+  |
       |                        |
       +-> Authenticated with   +-> Signed with
           Auth Tag N               Signature
```

Encrypted Frame with Signature

Note that the authentication tag for the current frame will only authenticate the SFrame header and the encrypted payload, ant not the signature nor the previous frames's authentication tags (N-1 to N-M) used to calculate the signature.

The last byte (NUM) after the authentication tag list and before the signature indicates the number of the authentication tags from previous frames present in the current frame. All the authentications tags MUST have the same size, which MUST be equal to the authentication tag size of the current frame. The signature is fixed size depending on the signature algorithm used (for example, 64 bytes for Ed25519).

The receiver has to keep track of all the frames received but yet not verified, by storing the authentication tags of each received frame. When a signature is received, the receiver will verify it with the signature key associated to the key id of the frame the signature was sent in. If the verification is successful, the received will mark the frames as authenticated and remove them from the list of the not verified frames. It is up to the application to decide what to do when signature verification fails.

When using SVC, the hash will be calculated over all the frames of the different spatial layers within the same superframe/picture. However the SFU will be able to drop frames within the same stream (either spatial or temporal) to match target bitrate.

If the signature is sent on a frame which layer that is dropped by the SFU, the receiver will not receive it and will not be able to perform the signature of the other received layers.

An easy way of solving the issue would be to perform signature only on the base layer or take into consideration the frame dependency graph and send multiple signatures in parallel (each for a branch of the dependency graph).

In case of simulcast or K-SVC, each spatial layer should be authenticated with different signatures to prevent the SFU to discard frames with the signature info.

In any case, it is possible that the frame with the signature is lost or the SFU drops it, so the receiver MUST be prepared to not receive a signature for a frame and remove it from the pending to be verified list after a timeout.

4.5. Ciphersuites

Each SFrame session uses a single ciphersuite that specifies the following primitives:

- A hash function used for key derivation and hashing signature inputs

- An AEAD encryption algorithm [RFC5116] used for frame encryption, optionally with a truncated authentication tag

- [Optional] A signature algorithm

This document defines the following ciphersuites:

Value	Name	Nk	Nn	Reference
0x0001	AES_CM_128_HMAC_SHA256_8	16	12	RFC XXXX
0x0002	AES_CM_128_HMAC_SHA256_4	16	12	RFC XXXX
0x0003	AES_GCM_128_SHA256	16	12	RFC XXXX
0x0004	AES_GCM_256_SHA512	32	12	RFC XXXX

In the "AES_CM" suites, the length of the authentication tag is indicated by the last value: "_8" indicates an eight-byte tag and "_4" indicates a four-byte tag.

In a session that uses multiple media streams, different ciphersuites might be configured for different media streams. For example, in order to conserve bandwidth, a session might use a ciphersuite with 80-bit tags for video frames and another ciphersuite with 32-bit tags for audio frames.

4.5.1. AES-CM with SHA2

In order to allow very short tag sizes, we define a synthetic AEAD function using the authenticated counter mode of AES together with HMAC for authentication. We use an encrypt-then-MAC approach as in SRTP [RFC3711].

Before encryption or decryption, encryption and authentication subkeys are derived from the single AEAD key using HKDF. The subkeys are derived as follows, where "Nk" represents the key size for the AES block cipher in use and "Nh" represents the output size of the hash function:

```
def derive_subkeys(key):
  aead_secret = HKDF-Extract(K, 'SFrame10 AES CM AEAD')
  enc_key = HKDF-Expand(aead_secret, 'enc', Nk)
  auth_key = HKDF-Expand(aead_secret, 'auth', Nh)
```

The AEAD encryption and decryption functions are then composed of individual calls to the CM encrypt function and HMAC. The resulting MAC value is truncated to a number of bytes "tag_len" fixed by the ciphersuite.

```
def compute_tag(aad, ct):
  aad_len = encode_big_endian(len(aad), 8)
  auth_data = aad_len + aad + ct
  tag = HMAC(auth_key, auth_data)
  return truncate(tag, tag_len)

def AEAD.Encrypt(key, nonce, aad, pt):
  ct = AES-CM.Encrypt(key, nonce, pt)
  tag = compute_tag(aad, ct)
  return ct + tag

def AEAD.Decrypt(key, nonce, aad, ct):
  inner_ct, tag = split_ct(ct, tag_len)

  candidate_tag = compute_tag(aad, inner_ct)
  if !constant_time_equal(tag, candidate_tag):
    raise Exception("Authentication Failure")

  return AES-CM.Decrypt(key, nonce, inner_ct)
```

5. Key Management

SFrame must be integrated with an E2E key management framework to exchange and rotate the keys used for SFrame encryption and/or signing. The key management framework provides the following functions:

- Provisioning KID/"base_key" mappings to participating clients

- (optional) Provisioning clients with a list of trusted signing keys

- Updating the above data as clients join or leave

It is up to the application to define a rotation schedule for keys. For example, one application might have an ephemeral group for every call and keep rotating key when end points joins or leave the call, while another application could have a persistent group that can be used for multiple calls and simply derives ephemeral symmetric keys for a specific call.

5.1. Sender Keys

If the participants in a call have a pre-existing E2E-secure channel, they can use it to distribute SFrame keys. Each client participating in a call generates a fresh encryption key and optionally a signing key pair. The client then uses the E2E-secure channel to send their encryption key and signing public key to the other participants.

In this scheme, it is assumed that receivers have a signal outside of SFrame for which client has sent a given frame, for example the RTP SSRC. SFrame KID values are then used to distinguish generations of the sender's key. At the beginning of a call, each sender encrypts with KID=0. Thereafter, the sender can ratchet their key forward for forward secrecy:

```
sender_key[i+1] = HKDF-Expand(
                    HKDF-Extract(sender_key[i], 'SFrame10 ratchet'),
                      '', AEAD.Nk)
```

The sender signals such an update by incrementing their KID value. A receiver who receives from a sender with a new KID computes the new key as above. The old key may be kept for some time to allow for out-of-order delivery, but should be deleted promptly.

If a new participant joins mid-call, they will need to receive from each sender (a) the current sender key for that sender, (b) the signing key for the sender, if used, and (c) the current KID value for the sender. Evicting a participant requires each sender to send a fresh sender key to all receivers.

5.2. MLS

The Messaging Layer Security (MLS) protocol provides group authenticated key exchange [I-D.ietf-mls-architecture] [I-D.ietf-mls-protocol]. In principle, it could be used to instantiate the sender key scheme above, but it can also be used more efficiently directly.

MLS creates a linear sequence of keys, each of which is shared among the members of a group at a given point in time. When a member joins or leaves the group, a new key is produced that is known only to the augmented or reduced group. Each step in the lifetime of the group is know as an "epoch", and each member of the group is assigned an "index" that is constant for the time they are in the group.

In SFrame, we derive per-sender "base_key" values from the group secret for an epoch, and use the KID field to signal the epoch and sender index. First, we use the MLS exporter to compute a shared SFrame secret for the epoch.

```
sframe_epoch_secret = MLS-Exporter("SFrame 10 MLS", "", AEAD.Nk)

sender_base_key[index] = HKDF-Expand(sframe_epoch_secret,
                           encode_big_endian(index, 4), AEAD.Nk)
```

For compactness, do not send the whole epoch number. Instead, we send only its low-order E bits. Note that E effectively defines a re-ordering window, since no more than 2^E epoch can be active at a given time. Receivers MUST be prepared for the epoch counter to roll over, removing an old epoch when a new epoch with the same E lower bits is introduced. (Sender indices cannot be similarly compressed.)

```
KID = (sender_index << E) + (epoch % (1 << E))
```

Once an SFrame stack has been provisioned with the "sframe_epoch_secret" for an epoch, it can compute the required KIDs and "sender_base_key" values on demand, as it needs to encrypt/ decrypt for a given member.

```
         ...
          |
Epoch 17 +--+-- index=33 -> KID = 0x211
          |  |
          |  +-- index=51 -> KID = 0x331
          |
          |
Epoch 16 +--+-- index=2 --> KID = 0x20
          |
          |
Epoch 15 +--+-- index=3 --> KID = 0x3f
          |  |
          |  +-- index=5 --> KID = 0x5f
          |
          |
Epoch 14 +--+-- index=3 --> KID = 0x3e
          |  |
          |  +-- index=7 --> KID = 0x7e
          |  |
          |  +-- index=20 -> KID = 0x14e
          |
         ...
```

MLS also provides an authenticated signing key pair for each participant. When SFrame uses signatures, these are the keys used to generate SFrame signatures.

6. Media Considerations

6.1. SFU

Selective Forwarding Units (SFUs) as described in https://tools.ietf.org/html/rfc7667#section-3.7 receives the RTP streams from each participant and selects which ones should be forwarded to each of the other participants. There are several approaches about how to do this stream selection but in general, in order to do so, the SFU needs to access metadata associated to each frame and modify the RTP information of the incoming packets when they are transmitted to the received participants.

This section describes how this normal SFU modes of operation interacts with the E2EE provided by SFrame

6.1.1. LastN and RTP stream reuse

The SFU may choose to send only a certain number of streams based on the voice activity of the participants. To reduce the number of SDP O/A required to establish a new RTP stream, the SFU may decide to reuse previously existing RTP sessions or even pre-allocate a predefined number of RTP streams and choose in each moment in time which participant media will be sending through it. This means that in the same RTP stream (defined by either SSRC or MID) may carry media from different streams of different participants. As different keys are used by each participant for encoding their media, the receiver will be able to verify which is the sender of the media coming within the RTP stream at any given point if time, preventing the SFU trying to impersonate any of the participants with another participant’s media. Note that in order to prevent impersonation by a malicious participant (not the SFU) usage of the signature is required. In case of video, the a new signature should be started each time a key frame is sent to allow the receiver to identify the source faster after a switch.

6.1.2. Simulcast

When using simulcast, the same input image will produce N different encoded frames (one per simulcast layer) which would be processed independently by the frame encryptor and assigned an unique counter for each.

6.1.3. SVC

In both temporal and spatial scalability, the SFU may choose to drop layers in order to match a certain bitrate or forward specific media sizes or frames per second. In order to support it, the sender MUST encode each spatial layer of a given picture in a different frame. That is, an RTP frame may contain more than one SFrame encrypted frame with an incrementing frame counter.

6.2. Video Key Frames

Forward and Post-Compromise Security requires that the e2ee keys are updated anytime a participant joins/leave the call.

The key exchange happens async and on a different path than the SFU signaling and media. So it may happen that when a new participant joins the call and the SFU side requests a key frame, the sender generates the e2ee encrypted frame with a key not known by the receiver, so it will be discarded. When the sender updates his sending key with the new key, it will send it in a non-key frame, so the receiver will be able to decrypt it, but not decode it.

Receiver will re-request an key frame then, but due to sender and sfu policies, that new key frame could take some time to be generated.

If the sender sends a key frame when the new e2ee key is in use, the time required for the new participant to display the video is minimized.

6.3. Partial Decoding

Some codes support partial decoding, where it can decrypt individual packets without waiting for the full frame to arrive, with SFrame this won't be possible because the decoder will not access the packets until the entire frame is arrived and decrypted.

7. Overhead

The encryption overhead will vary between audio and video streams, because in audio each packet is considered a separate frame, so it will always have extra MAC and IV, however a video frame usually consists of multiple RTP packets. The number of bytes overhead per frame is calculated as the following 1 + FrameCounter length + 4 The constant 1 is the SFrame header byte and 4 bytes for the HBH authentication tag for both audio and video packets.

7.1. Audio

Using three different audio frame durations 20ms (50 packets/s) 40ms (25 packets/s) 100ms (10 packets/s) Up to 3 bytes frame counter (3.8 days of data for 20ms frame duration) and 4 bytes fixed MAC length.

Counter len	Packets	Overhead	Overhead	Overhead
		bps@20ms	bps@40ms	bps@100ms
1	0-255	2400	1200	480
2	255 - 65K	2800	1400	560
3	65K - 16M	3200	1600	640

7.2. Video

The per-stream overhead bits per second as calculated for the following video encodings: 30fps@1000Kbps (4 packets per frame) 30fps@512Kbps (2 packets per frame) 15fps@200Kbps (2 packets per frame) 7.5fps@30Kbps (1 packet per frame) Overhead bps = (Counter length + 1 + 4) * 8 * fps

Counter len	Frames	Overhead	Overhead	Overhead
		bps@30fps	bps@15fps	bps@7.5fps
1	0-255	1440	1440	720
2	256 - 65K	1680	1680	840
3	56K - 16M	1920	1920	960
4	16M - 4B	2160	2160	1080

7.3. SFrame vs PERC-lite

[RFC8723] has significant overhead over SFrame because the overhead is per packet, not per frame, and OHB (Original Header Block) which duplicates any RTP header/extension field modified by the SFU. [I-D.murillo-perc-lite] <https://mailarchive.ietf.org/arch/msg/perc/SB0qMHWz6EsDtz3yIEX0HWp5IEY/> is slightly better because it doesn't use the OHB anymore, however it still does per packet encryption using SRTP. Below the the overheard in [I-D.murillo-perc-lite] implemented by Cosmos Software which uses extra 11 bytes per packet to preserve the PT, SEQ_NUM, TIME_STAMP and SSRC fields in addition to the extra MAC tag per packet.

OverheadPerPacket = 11 + MAC length Overhead bps = PacketPerSecond * OverHeadPerPacket * 8

Similar to SFrame, we will assume the HBH authentication tag length will always be 4 bytes for audio and video even though it is not the case in this [I-D.murillo-perc-lite] implementation

7.3.1. Audio

Overhead bps@20ms	Overhead bps@40ms	Overhead bps@100ms
6000	3000	1200

7.3.2. Video

Overhead bps@30fps	Overhead bps@15fps	Overhead bps@7.5fps
(4 packets per frame) 14400	(2 packets per frame) 7200	(1 packet per frame) 3600

For a conference with a single incoming audio stream (@ 50 pps) and 4 incoming video streams (@200 Kbps), the savings in overhead is 34800 - 9600 = ~25 Kbps, or ~3%.

8. Security Considerations

8.1. Key Management

Key exchange mechanism is out of scope of this document, however every client MUST change their keys when new clients joins or leaves the call for "Forward Secrecy" and "Post Compromise Security".

8.2. Authentication tag length

The cipher suites defined in this draft use short authentication tags for encryption, however it can easily support other ciphers with full authentication tag if the short ones are proved insecure.

9. IANA Considerations

This document makes no requests of IANA.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

10.2. Informative References

[I-D.ietf-mls-architecture] Omara, E., Beurdouche, B., Rescorla, E., Inguva, S., Kwon, A., and A. Duric, "The Messaging Layer Security (MLS) Architecture", draft-ietf-mls-architecture-05 (work in progress), July 2020.

[I-D.ietf-mls-protocol] Barnes, R., Beurdouche, B., Millican, J., Omara, E., Cohn-Gordon, K., and R. Robert, "The Messaging Layer Security (MLS) Protocol", draft-ietf-mls-protocol-10 (work in progress), October 2020.

[I-D.murillo-perc-lite] Murillo, S. and A. Gouaillard, "End to End Media Encryption Procedures", draft-murillo-perc-lite-01 (work in progress), May 2020.

[RFC3711] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", RFC 3711, DOI 10.17487/RFC3711, March 2004, <https://www.rfc-editor.org/info/rfc3711>.

[RFC8723] Jennings, C., Jones, P., Barnes, R., and A. Roach, "Double Encryption Procedures for the Secure Real-Time Transport Protocol (SRTP)", RFC 8723, DOI 10.17487/RFC8723, April 2020, <https://www.rfc-editor.org/info/rfc8723>.

Authors' Addresses

Emad Omara
Google

Email: emadomara@google.com

Justin Uberti
Google

Email: juberti@google.com

Alexandre GOUAILLARD
CoSMo Software

Email: Alex.GOUAILLARD@cosmosoftware.io

Sergio Garcia Murillo
CoSMo Software

Email: sergio.garcia.murillo@cosmosoftware.io