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

R. Barnes
Cisco
R. Robert
Wire

16 November 2020

Using Messaging Layer Security (MLS) to Provide Keys for SFrame
draft-barnes-sframe-mls-00

Abstract

Secure Frames (SFrame) defines a compact scheme for encrypting real-time media. In order for SFrame to address cases where media are exchanged among many participants (e.g., real-time conferencing), it needs to be augmented with a group key management protocol. The Messaging Layer Security (MLS) protocol provides continuous group authenticated key exchange, allowing a group of participants in a media session to authenticate each other and agree on a group key. This document defines how the group keys produced by MLS can be used with SFrame to secure real-time sessions for groups.

Discussion Venues

This note is to be removed before publishing as an RFC.

Source for this draft and an issue tracker can be found at https://github.com/bifurcation/sframe-mls.

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 20 May 2021.

Table of Contents

1. Introduction

Secure Frames (SFrame) defines a compact scheme for encrypting real-time media. In order for SFrame to address cases where media are exchanged among many participants (e.g., real-time conferencing), it needs to be augmented with a group key management protocol. The Messaging Layer Security (MLS) protocol [!I-D.ietf-mls-protocol] provides continuous group authenticated key exchange. MLS provides several important security properties [!I-D.ietf-mls-arch]:

* Group Key Exchange: All members of the group at a given time know a secret key that is inaccessible to parties outside the group.

* Authentication of group members: Each member of the group can authenticate the other members of the group.

* Group Agreement: The members of the group all agree on the identities of the participants in the group.

* Forward Secrecy: There are protocol events such that if a member's state is compromised after the event, group secrets created before the event are safe.

* Post-compromise Security: There are protocol events such that if a member's state is compromised before the event, the group secrets created after the event are safe.

When a real-time session uses MLS as the basis for SFrame keys, these security properties apply to real-time media as well. In the remainder of this document, we define how to use the secrets produced by MLS to generate the keys required by SFrame.

[[OPEN ISSUE: We could define an MLS extension that would provide negotiation of SFrame parameters, notably the ciphersuite and the value E defined below.]]

2. SFrame Key Management

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)
```

[[OPEN ISSUE: MLS has its own "secret tree" that provides better forward secrecy properties within an epoch. (This scheme provides none.) An alternative approach would be to re-use the MLS secret tree, either directly or as a data structure.]]

The Key ID (KID) field in the SFrame header provides the epoch and index values that are needed to generate the appropriate key from the MLS key schedule.

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

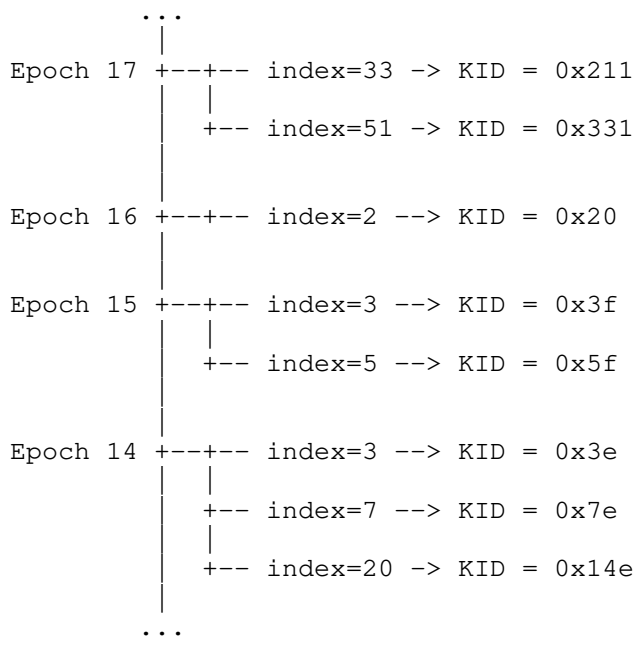
For compactness, do not send the whole epoch number. Instead, we send only its low-order E bits. The participants in the group MUST agree on the value of E for a given session, through some negotiation not specified here.

Note that E effectively defines a re-ordering window, since no more than 2^E epoch can be active at a given time. The better the participants are in sync with regard to key roll-over, and the less reordering of SFrame-protected payloads by the network, the fewer bits of epoch are necessary.

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.

[[OPEN ISSUE: There might be some considerations for new joiners. Some trial decryption might be necessary to detect whether you're in epoch N or in epoch N + 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.



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

3. Security Considerations

The security properties provided by MLS are discussed in detail in [!I-D.ietf-mls-arch] and [!I-D.ietf-mls-protocol]. This document extends those guarantees to SFrame.

It should be noted that the per-sender keys derived here do not provide per-sender authentication, since any member of the group could derive the same keys (as indeed they must in order to decrypt the protected payload). Per-sender keys are derived only to avoid nonce collision among multiple unsynchronized senders. So the authentication limitations of SFrame remain: There is per-sender authentication only when signatures are used. Otherwise, SFrame only authenticates membership in the group, and members are free to impersonate each other.

4. IANA Considerations

This document makes no request of IANA.

Appendix A. Acknowledgements

TODO

Authors' Addresses

Richard Barnes
Cisco

Email: rlb@ipv.sx

Raphael Robert
Wire

Email: raphael@wire.com

AVTCORE
Internet-Draft
Intended status: Standards Track
Expires: September 9, 2021

S. Garcia Murillo
CoSMo Software
Y. Fablet
Apple Inc.
A. Gouaillard
CoSMo Software

March 08, 2021

Codec agnostic RTP payload format for video
draft-gouaillard-avtcore-codec-agn-rtp-payload-01

Abstract

RTP Media Chains usually rely on piping encoder output directly to packetizers. Media packetization formats often support a specific codec format and optimize RTP packets generation accordingly.

With the development of Selective Forward Unit (SFU) solutions, that do not process media content server side, the need for media content processing at the origin and at the destination has arised.

RTP Media Chains used e.g. in WebRTC solutions are increasingly relying on application-specific transforms that sit in-between encoder and packetizer on one end and in-between depacketizer and decoder on the other end. This use case has become so important, that the W3C is standardizing the capacity to access encoded content with the [WebRTCInsertableStreams] API proposal. An extremely popular use case is application level end-to-end encryption of media content, using for instance [SFrame].

Whatever the modification applied to the media content, RTP packetizers can no longer expect to use packetization formats that mandate media content to be in a specific codec format.

In the extreme cases like encryption, where the RTP Payload is made completely opaque to the SFUs, some extra mechanism must also be added for them to be able to route the packets without depending on RTP payload or payload headers.

The traditionnal process of creating a new RTP Payload specification per content would not be practical as we would need to make a new one for each codec-transform pair.

This document describes a solution, which provides the following features in the case the encoded content has been modified before reaching the packetizer: - a paylaod agnostic RTP packetization format that can be used on any media content, - a negotiation

mechanism for the above format and the inner payload, Both of the above mechanism are backward compatible with most of (S)RTP/RTCP mechanisms used for bandwidth estimation and congestion control in RTP/SRTP/webrtc, including but not limited to SSRC, RED, FEC, RTX, NACK, SR/RR, REMB, transport-wide-CC, TMBR, ... It as illustrated by existing implementations in chrome, safari, and Medooze.

This document also describes a solution to allow SFUs to continue performing packet routing on top of this generic RTP packetization format.

This document complements the SFrame (media encryption), and Dependency Descriptor (AV1 payload annex) documents to provide an End-to-End-Encryption solution that would sit on top of SRTP/Webrtc, use SFUs on the media back-end, and leverage W3C APIs in the browser. A high level description of such system will be provided as an informational I-D in the SFrame WG and then cited here.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

As per Figure 1 of [RFC7656], a Media Packetizer transforms a single Encoded Stream into one or several RTP packets. The Encoded Stream is coming straight from the Media Encoder and is expected to follow the format produced by the Media Encoder. A number of Media Packetizer formats have been designed to process a specific format produced by Media Encoder. For instance [RFC6184] is dedicated to the processing of content produced by H.264 Media Encoders, and generates packets following NALUs organization.

WebRTC applications are increasingly deploying end-to-end encryption solutions on top of RTP Media Chains. End-to-end encryption is implemented by inserting application-specific Media Transformers between Media Encoder and Media Packetizer on the sending side, and between Media Depacketizer and Media Decoder on the receiving side, as described in Figure 1 and Figure 2. To support end-to-end encryption, Media Transformers can use the [SFrame] format. In browsers, Media Transformers are implemented using

[WebRTCInsertableStreams], for instance by injecting JavaScript code provided by web pages.

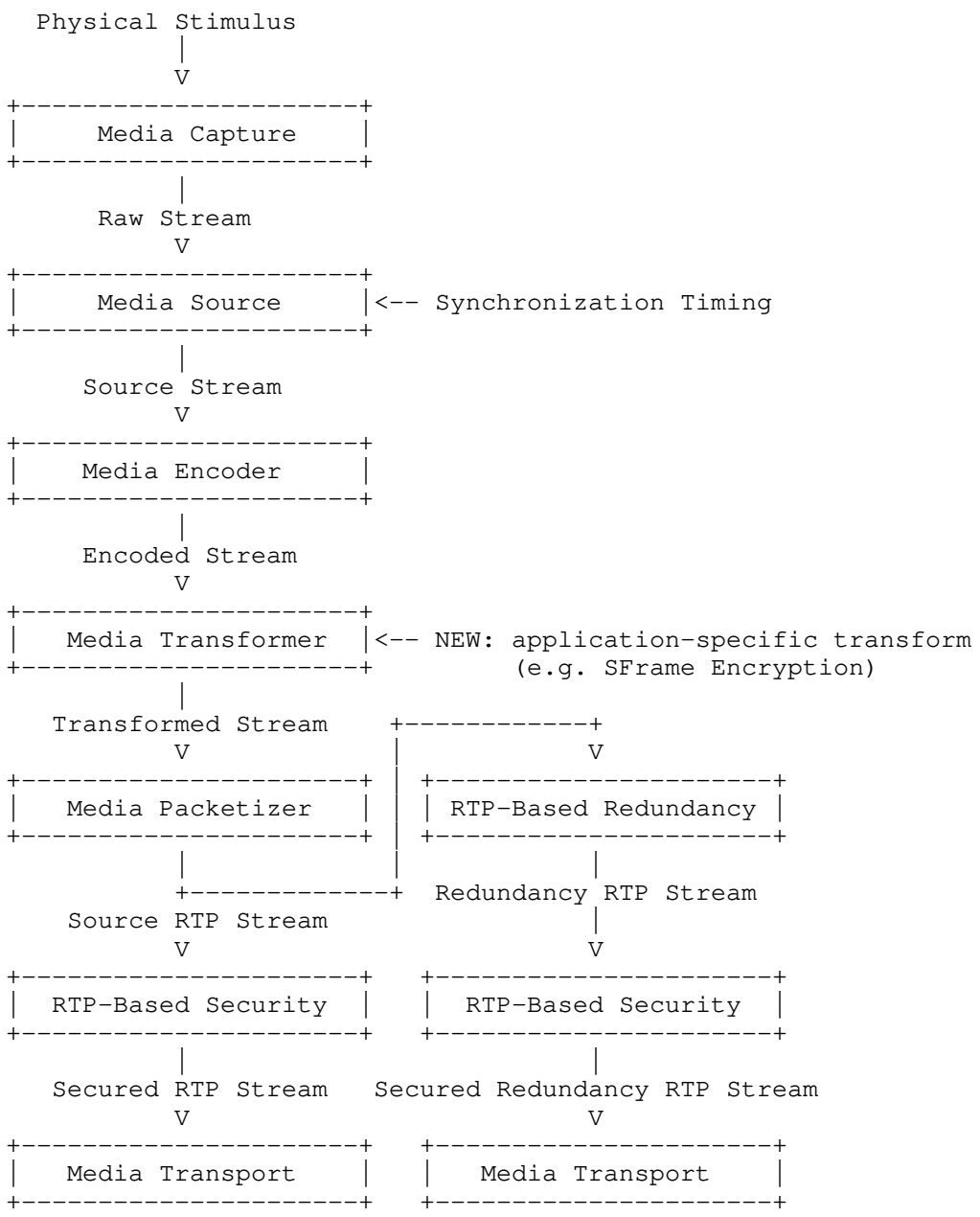
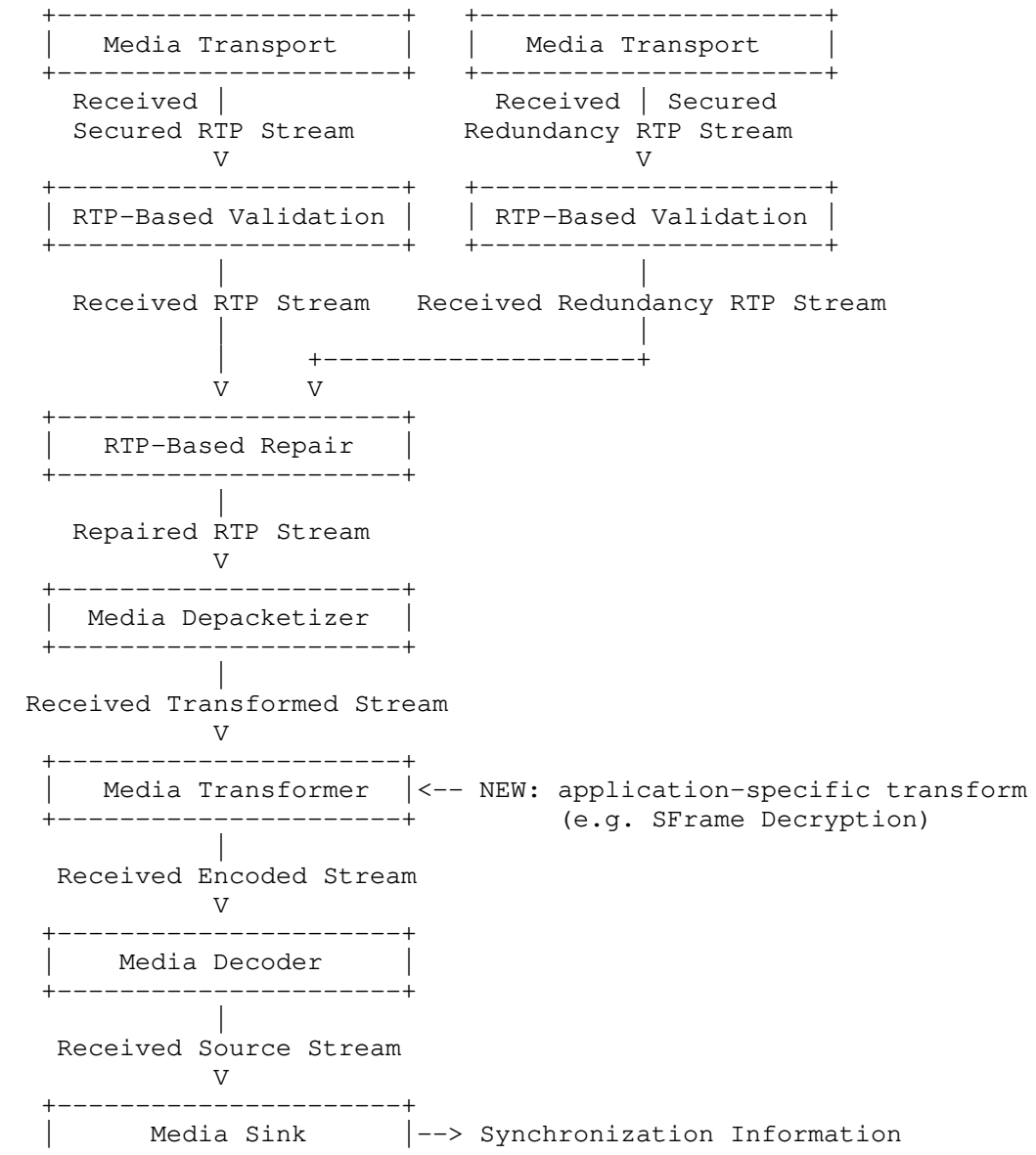


Figure 1: Sender Side Concepts in the Media Chain
With Application-level Media Transform

These RTP packets are sent over the wire to a receiver media chain matching the sender side, reaching the Media Depacketizer that will reconstruct the Encoded Stream before passing it to the Media Decoder.



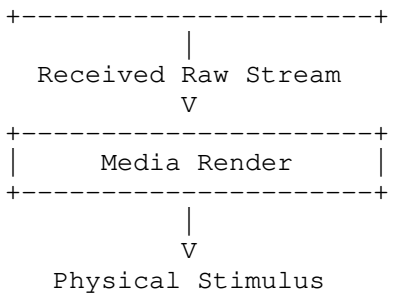


Figure 2: Receiver Side Concepts in the Media Chain With Application-level Media Transform

This generic packetization does not change how the mapping between one or several encoded or dependant streams are mapped to the RTP streams or how the synchronization sources(s) (SSRC) are assigned.

Given the use of post-encoder application-specific transforms, the whole Media Chain needs to be made aware of it. This includes the sender post-transform Media Chain, Media Transport intermediaries (SFUs typically) and receiver pre-transform Media Chain.

As these transforms can alter Encoded Streams in any possible way, the use of codec-specific Media Packetizers like [RFC6184] on Transformed Stream may be suboptimal on sender side. It may also be problematic on the receiving side in case codec-specific processing is done prior the Media Transformer. Media Transport intermediaries are often looking at the Media Content itself to fuel their packet selection algorithms.

2. Goals

The objective of this document is to support inserting any application-specific transform between encoders and packetizers in the Media Chain. For that purpose, this document will: 1. Provide a generic packetization format that supports any media content (compressed audio, compressed video, encrypted content...) that allows reuse of existing RTP mechanisms in place in WebRTC applications such as RTX, RED or FEC. 2. Provide a way to negotiate use of the generic packetization format between sender and receiver, with minimum impact on existing negotiation approaches. 3. Provide a side-channel information so that network intermediaries (SFU in particular) can do their existing packet routing strategies without inspecting the media content.

3. RTP Packetization

A generic packetizer, by design, is not expected to understand the format of the media to transmit. The unit used by the packetizer to do processing is called a frame in the remainder of the document.

It is the responsibility of the application using the packetizer to group media content in meaningful frames. In the common case of a video codec, the packetizer frame is the frame in byte format (h264 annex b for example) generated by the encoder.

If the application wants to transform encoded content, the application needs to split the encoded content into frames prior the transform. Each frame is then transformed independently, for instance encrypted using [SFrame]. The content of each transformed frame is then processed by the packetizer.

In the case of a video codec supporting spatial scalability, each spatial layer MUST be split in its own frame by the application before passing it to the packetizer.

When the packetizer receives a frame from the application, it MUST fragment the frame content in multiple RTP packets to ensure packets do not exceed the network maximum transmission unit. The content of the frame will be treated as a binary blob by the packetizer, so the decision about the boundaries of each fragment is decided arbitrarily by the packetizer. The packetizer or any relaying server MUST NOT modify the frame content and concatenating the RTP payload of the RTP packets for each frame MUST produce the exact binary content of the input frame content.

The marker bit of each RTP packet in a frame MUST be set according to the audio and video profiles specified in [RFC3551].

The spatial layer frames are sent in ascending order, with the same RTP timestamp, and only the last RTP packet of the last spatial layer frame will have the marker bit set to 1.

4. Payload Multiplexing

In order to reduce the number of payload type in the SDP exchange, a single payload type code for the generic packetization can be used for all negotiated media formats. That requires to identify the original payload type code of the frame negotiated media format, called the associated payload type (APT) hereunder. The APT value is the payload type code of the associated format passed to the generic Media Packetizer before any transformation is applied.

The APT value is sent in a dedicated header extension. The payload of this header extension can be encoded using either the one-byte or two-byte header defined in [RFC5285]. Figures 3 and 4 show examples with each one of these examples.

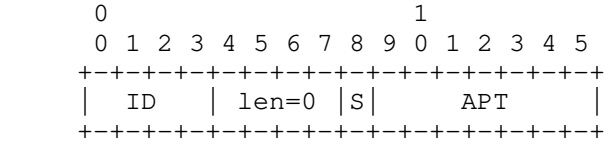


Figure 3: Frame Associated Payload Type Encoding Using the One-Byte Header Format

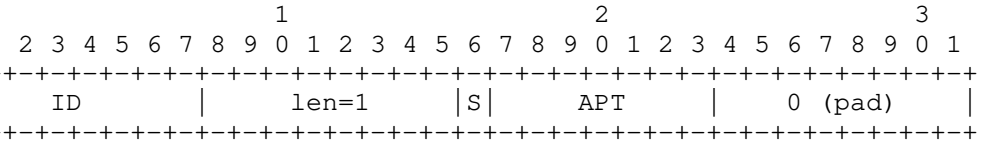


Figure 4: Frame Associated Payload Type Encoding Using the Two-Byte Header Format

The APT value is the associated payload type value. The S bit indicates if the media stream can be forwarded safely starting from this RTP packet. Typically, it will be set to 1 on the first RTP packet of an intra video frame and in all RTP audio packets.

Receivers MUST be ready to receive RTP packets with different associated payload types in the same way they would receive different payload type codes on the RTP packets.

The URI for declaring this header extension in an extmap attribute is "urn:ietf:params:rtp-hdrext:associated-payload-type".

5. SDP Negotiation

To use the RTP generic packetization, the SDP Offer/Answer exchange MUST negotiate: - The payload type of the negotiated codec format - The generic payload type - The associated payload type header extension

Only the negotiated payload types are allowed to be used as associated payload types. Figure 5 illustrates a SDP that negotiates exchange of video using either VP8 or VP9 codecs with the possibility to use the generic packetization. In this example, RTX is also negotiated and will be applied normally on each associated payload type.

```
m=video 9 UDP/TLS/RTP/SAVPF 96 97 98 99 100 101
c=IN IP4 0.0.0.0
a=rtcp:9 IN IP4 0.0.0.0
a=setup:actpass
a=mid:1
a=extmap:1 urn:ietf:params:rtp-hdrext:sdes:mid
a=extmap:2 urn:ietf:params:rtp-hdrext:sdes:rtp-stream-id
a=extmap:3 urn:ietf:params:rtp-hdrext:sdes:repaired-rtp-stream-id
a=extmap:4 urn:ietf:params:rtp-hdrext:associated-payload-type
a=sendrecv
a=rtpmap:96 vp9/90000
a=rtpmap:97 vp8/90000
a=rtpmap:98 generic/90000
a=rtpmap:99 rtx/90000
a=fmtp:99 apt=96
a=rtpmap:100 rtx/90000
a=fmtp:100 apt=97
a=rtpmap:101 rtx/90000
a=fmtp:101 apt=98
```

Figure 5: SDP example negotiating the generic payload type and related header extension for video

6. SFU Packet Selection

SFUs need to have a basic understanding of each frame they receive so they can decide to forward it or not and to which endpoint. They might need similar information to support media content recording. This information is either generic to a group of frame (called a stream hereafter) or specific to each frame.

The information is transmitted as a RTP header extension as the RTP packet payload should be treated as opaque by the SFU. This is especially necessary if the payload is end-to-end encrypted. The amount of information should be limited to what is strictly necessary to the SFU task since it is not always as trusted as individual peers.

For audio, configuration information such as Opus TOC might be useful. For video, configuration information might include: - Stream configuration information: resolution, quality, frame rate... - Codec specific configuration information: codec profile like profile_idc... - Frame specific information: whether the stream is decodable when starting from this frame, whether the frame is skippable...

For video content, this information can be sent using a Dependency Descriptor header extension. In that case, the first RTP packet of

the frame will have its start_of_frame equal to 1 and the last packet will have its end_of_frame equal to 1.

7. Redundancy Techniques Considerations

The solution described in this document is expected to integrate well with the existing RTP ecosystem. This section describes how the generic packetizer can be used jointly with existing techniques that allow to mitigate unreliable transports.

7.1. Retransmission Techniques

[RFC4588] defines a retransmission payload format (RTX) that can be used in case of packet loss. As defined in [RFC4588], RTX is able to handle any payload format, including the format described in this document. Given RTX preserves both RTP packet payload and headers, the receiver will be able to identify the payload type of the recovered packet and whether generic packetization is used. RTX will also allow recovering RTP header extensions that convey information on the media content itself.

7.2. Forward Error Correction (FEC) Techniques

FEC is another technique used in RTP Media Chains to protect media content against packet loss. [RFC5109] defines such a payload format used to transmit FEC for specific packets protection.

FEC may protect some parts of the media content more than others. For instance, intra video frame encoded data or important network abstraction layer units (NALUs) like SPS/PPS may be more protected. With a post-encoder transform and the use of a generic packetization, the granularity of the recovery mechanism is no longer at the NALU level but at the level of the frame generated by the post-encoder transform. In case a SVC codec is used, each spatial layer will be processed as an independent frame. In that case, base layers can be protected more heavily than higher resolution layers.

7.3. Redundant Audio Data Techniques

As defined in [RFC7656] RTP-based redundancy is defined here as a transformation that generates redundant or repair packets sent out as a Redundancy RTP Stream to mitigate Network Transport impairments, like packet loss and delay.

[RFC2198] defines a payload format for sending the same audio data encoded multiple times at different quality levels. This allows to use a lower quality encoding of the audio data, should the higher quality encoding of the audio data is lost during the transmission.

If a Media Transformation is in use, both the primary and redundant encoding must be transformed independently and the redundant packet created normally. As the RTP headers present in the redundant packet are only applicable to the primary encoding, if the payload type for a redundant encoding block is mapped to the generic packetizer, the value of the associated payload type for the primary encoding is applied to the redundant encoding block as well.

8. Alternatives

Various alternatives can be used to implement and negotiate generic packetization. This section describes a few additional alternatives. This section is to be removed before finalization of the document.

8.1. Generic Packetization With In-Payload APT

Instead of using a RTP header extension to convey the APT value, it is prepended in the RTP payload itself. As the value cannot change for a whole frame, its value is prepended to the first packet generated of the frame only. This removes the need to negotiate a dedicated header extension, but may require the SFU to update the payload when sending or recording content.

8.2. A Payload Type for Generic Packetization AND Media Format

The payload type is negotiated in the SDP so as to identify both the negotiated codec format and the generic packetization use. There is no network cost but this increases the number of payload types used in the SDP.

```
m=video 9 UDP/TLS/RTP/SAVPF 96 97 98 99 100 101
c=IN IP4 0.0.0.0
a=rtcp:9 IN IP4 0.0.0.0
a=setup:actpass
a=mid:1
a=extmap:1 urn:ietf:params:rtp-hdrext:sdes:mid
a=extmap:2 urn:ietf:params:rtp-hdrext:sdes:rtp-stream-id
a=extmap:3 urn:ietf:params:rtp-hdrext:sdes:repaired-rtp-stream-id
a=sendrecv
a=rtpmap:96 vp9/90000
a=rtpmap:97 generic/90000
a=fmtp:97 apt=96
a=rtpmap:98 vp8/90000
a=rtpmap:99 generic/90000
a=fmtp:99 apt=98
a=rtpmap:100 rtx/90000
a=fmtp:100 apt=96
a=rtpmap:101 rtx/90000
a=fmtp:101 apt=97
a=rtpmap:102 rtx/90000
a=fmtp:102 apt=98
a=rtpmap:103 rtx/90000
a=fmtp:103 apt=99
```

Figure 6: SDP example negotiating a payload type for format and generic packetization

A variation of this approach is to consider defining generic payload types, each of them having an identified codec format.

```
m=video 9 UDP/TLS/RTP/SAVPF 96 97 98 99 100 101
c=IN IP4 0.0.0.0
a=rtcp:9 IN IP4 0.0.0.0
a=setup:actpass
a=mid:1
a=extmap:1 urn:ietf:params:rtp-hdrext:sdes:mid
a=extmap:2 urn:ietf:params:rtp-hdrext:sdes:rtp-stream-id
a=extmap:3 urn:ietf:params:rtp-hdrext:sdes:repaired-rtp-stream-id
a=sendrecv
a=rtpmap:96 generic/90000
a=fmtp:96 codec=vp9
a=rtpmap:97 generic/90000
a=fmtp:97 codec=vp8
a=rtpmap:98 rtx/90000
a=fmtp:98 apt=96
a=rtpmap:99 rtx/90000
a=fmtp:99 apt=97
```

Figure 7: SDP example negotiating a payload type for format and generic packetization

8.3. A RTP Header To Choose Packetization

A RTP header extension can be used to flag content as opaque so that the receiver knows whether to use or not the generic packetization. As for the API header extension, the RTP header extension may not need to be sent for every packet, it could for instance be sent for the first packet of every intra video frame. The main advantage of this approach is the reduced impact on SDP negotiation.

```
m=video 9 UDP/TLS/RTP/SAVPF 96 97 98 99 100 101
c=IN IP4 0.0.0.0
a=rtcp:9 IN IP4 0.0.0.0
a=setup:actpass
a=mid:1
a=extmap:1 urn:ietf:params:rtp-hdrext:sdes:mid
a=extmap:2 urn:ietf:params:rtp-hdrext:sdes:rtp-stream-id
a=extmap:3 urn:ietf:params:rtp-hdrext:sdes:repaired-rtp-stream-id
a=extmap:4 urn:ietf:params:rtp-hdrext:generic-packetization-use
a=sendrecv
a=rtpmap:96 vp9/90000
a=rtpmap:97 vp8/90000
a=rtpmap:98 rtx/90000
a=fmtp:98 apt=96
a=rtpmap:99 rtx/90000
a=fmtp:99 apt=97
```

Figure 8: SDP example negotiating generic packetization as RTP header extension

9. Security Considerations

RTP packets using the payload format defined in this specification are subject to the general security considerations discussed in [RFC3550]. It is not expected that the proposed solutions (generic packetization and header extension) presented in this document can create new security threats. The use and implementation of RTP Media Chains containing Media Transformers needs to be done carerefully. It is important to refer to the security considerations discussed in [SFrame] and [WebRTCInsertableStreams]. In particular Media Transformers on the receiver side need to be prepared to receive arbitrary content, like decoders already do. Similarly, since Media Transformers can be implemented as JavaScript in browsers, RTP Packetizers should be prepared to receive arbitrary content.

10. IANA Considerations

Two new media subtypes have been registered with IANA, as described in this section.

10.1. Registration of audio/generic

Type name: audio

Subtype name: generic

Required parameters: none

Optional parameters: none

Encoding considerations: This format is framed (see Section 4.8 in the template document) and contains binary data.

Security considerations: TBD.

Interoperability considerations: TBD

Published specification: TBD.

Applications that use this media type: TBD.

Additional information: none

Intended usage: COMMON

Restrictions on usage: TBD

Author:

Change controller:

11. Registration of video/generic

Type name: video

Subtype name: generic

Required parameters: none

Optional parameters: none

Encoding considerations: This format is framed (see Section 4.8 in the template document) and contains binary data.

Security considerations: TBD.

Interoperability considerations: TBD

Published specification: TBD.

Applications that use this media type: TBD.

Additional information: none

Intended usage: COMMON

Restrictions on usage: TBD

Author:

Change controller:

12. References

12.1. 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>.

[RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", STD 64, RFC 3550, DOI 10.17487/RFC3550, July 2003, <https://www.rfc-editor.org/info/rfc3550>.

[RFC3551] Schulzrinne, H. and S. Casner, "RTP Profile for Audio and Video Conferences with Minimal Control", STD 65, RFC 3551, DOI 10.17487/RFC3551, July 2003, <https://www.rfc-editor.org/info/rfc3551>.

[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>.

[RFC4566] Handley, M., Jacobson, V., and C. Perkins, "SDP: Session Description Protocol", RFC 4566, DOI 10.17487/RFC4566, July 2006, <https://www.rfc-editor.org/info/rfc4566>.

[RFC5285] Singer, D. and H. Desineni, "A General Mechanism for RTP Header Extensions", RFC 5285, DOI 10.17487/RFC5285, July 2008, <https://www.rfc-editor.org/info/rfc5285>.

[RFC7656] Lennox, J., Gross, K., Nandakumar, S., Salgueiro, G., and B. Burman, Ed., "A Taxonomy of Semantics and Mechanisms for Real-Time Transport Protocol (RTP) Sources", RFC 7656, DOI 10.17487/RFC7656, November 2015, <https://www.rfc-editor.org/info/rfc7656>.

[RFC8285] Singer, D., Desineni, H., and R. Even, Ed., "A General Mechanism for RTP Header Extensions", RFC 8285, DOI 10.17487/RFC8285, October 2017, <https://www.rfc-editor.org/info/rfc8285>.

12.2. Informative References

[RFC2198] Perkins, C., Kouvelas, I., Hodson, O., Hardman, V., Handley, M., Bolot, J., Vega-Garcia, A., and S. Fosse-Parisis, "RTP Payload for Redundant Audio Data", RFC 2198, DOI 10.17487/RFC2198, September 1997, <https://www.rfc-editor.org/info/rfc2198>.

[RFC4588] Rey, J., Leon, D., Miyazaki, A., Varsa, V., and R. Hakenberg, "RTP Retransmission Payload Format", RFC 4588, DOI 10.17487/RFC4588, July 2006, <https://www.rfc-editor.org/info/rfc4588>.

[RFC5109] Li, A., Ed., "RTP Payload Format for Generic Forward Error Correction", RFC 5109, DOI 10.17487/RFC5109, December 2007, <https://www.rfc-editor.org/info/rfc5109>.

[RFC6184] Wang, Y., Even, R., Kristensen, T., and R. Jesup, "RTP Payload Format for H.264 Video", RFC 6184, DOI 10.17487/RFC6184, May 2011, <https://www.rfc-editor.org/info/rfc6184>.

[RFC6464] Lennox, J., Ed., Ivov, E., and E. Marocco, "A Real-time Transport Protocol (RTP) Header Extension for Client-to-Mixer Audio Level Indication", RFC 6464, DOI 10.17487/RFC6464, December 2011, <https://www.rfc-editor.org/info/rfc6464>.

[RFC6465] Ivov, E., Ed., Marocco, E., Ed., and J. Lennox, "A Real-time Transport Protocol (RTP) Header Extension for Mixer-to-Client Audio Level Indication", RFC 6465, DOI 10.17487/RFC6465, December 2011, <https://www.rfc-editor.org/info/rfc6465>.

[RFC6904] Lennox, J., "Encryption of Header Extensions in the Secure Real-time Transport Protocol (SRTP)", RFC 6904, DOI 10.17487/RFC6904, April 2013, <https://www.rfc-editor.org/info/rfc6904>.

[SFrame] "Secure Frame (SFrame)", n.d., <https://tools.ietf.org/html/draft-omara-sframe>.

[WebRTCInsertableStreams] "WebRTC Insertable Media using Streams", n.d., <https://w3c.github.io/webrtc-insertable-streams>.

Authors' Addresses

Sergio Garcia Murillo
CoSMo Software

Email: sergio.garcia.murillo@cosmosoftware.io

Youenn Fablet
Apple Inc.

Email: youenn@apple.com

Alexandre Gouaillard
CoSMo Software

Email: alex.gouaillard@cosmosoftware.io

Network Working Group
Internet-Draft
Intended status: Informational
Expires: November 20, 2020

E. Omara
J. Uberti
Google
A. GOUAILLARD
S. Murillo
CoSMo Software

May 19, 2020

Secure Frame (SFrame)
draft-omara-sframe-00

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 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 November 20, 2020.

Copyright Notice

Table of Contents

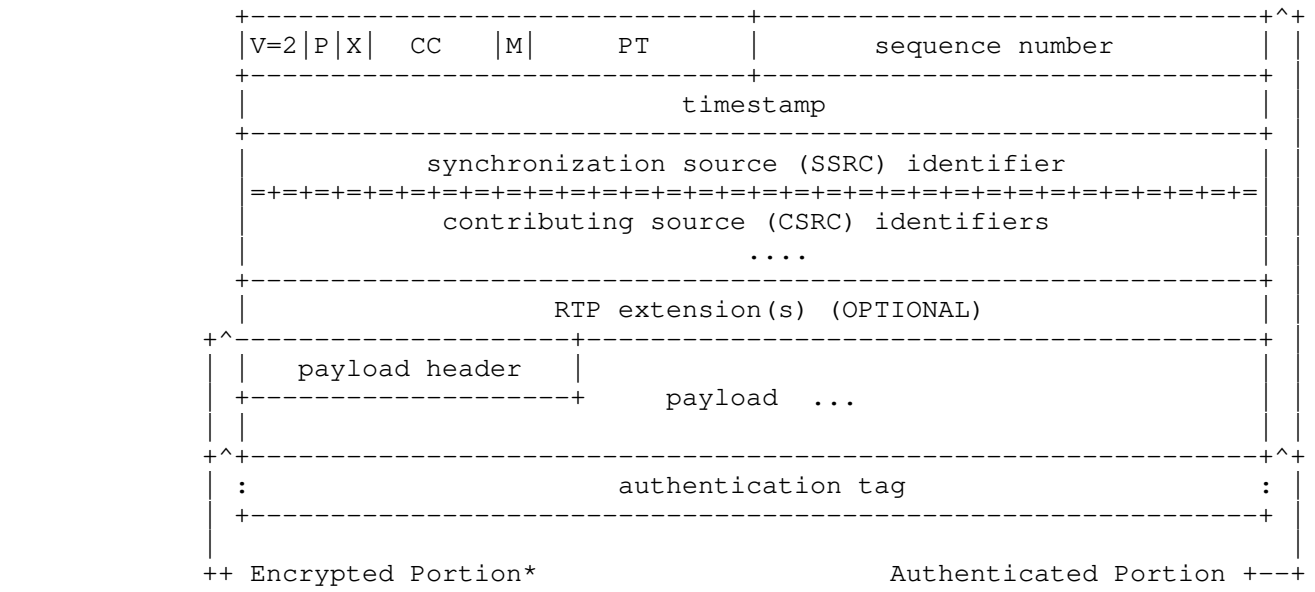
1. Introduction

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.



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.

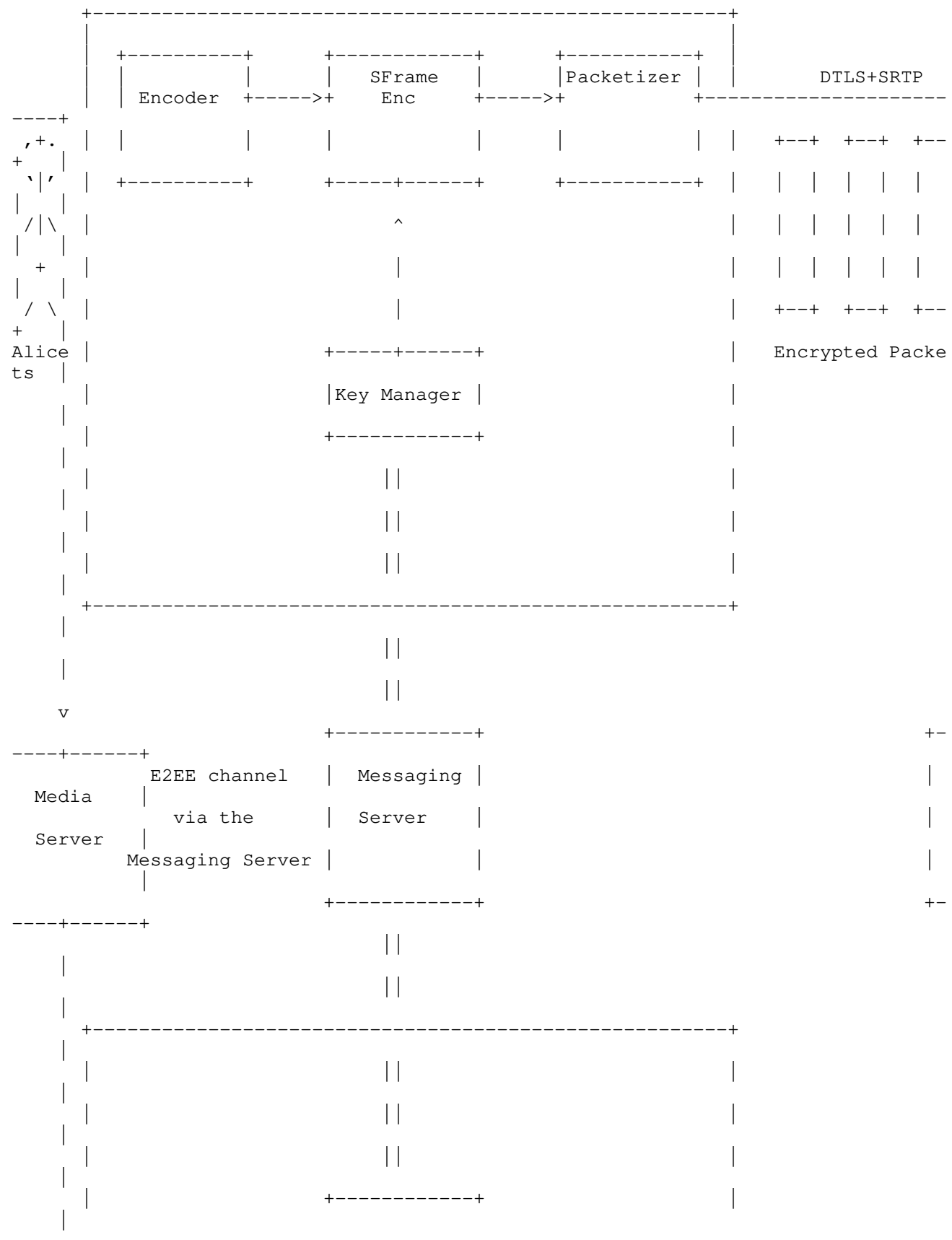
8. Work with the most popular audio and video codecs used in conferencing scenarios.

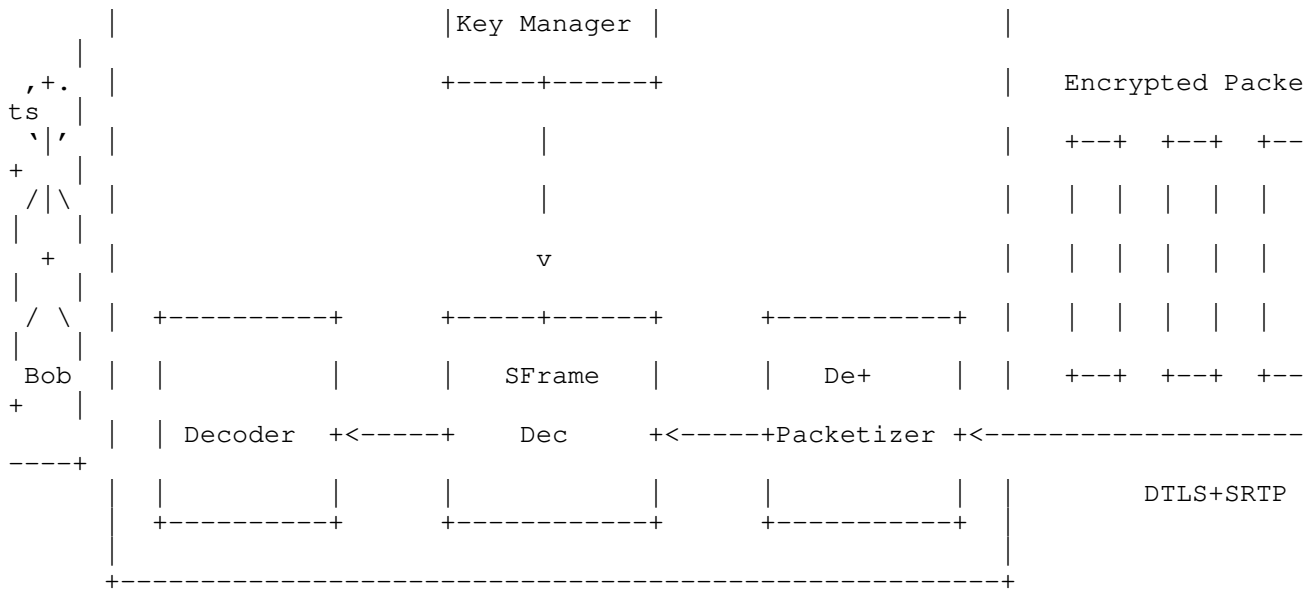
4. SFrame

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.

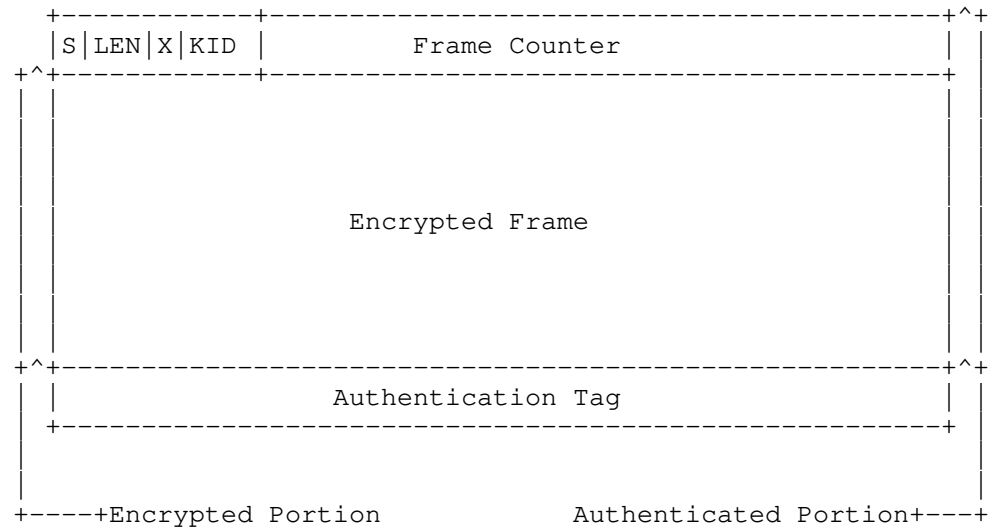
The generic packetizer splits the E2E encrypted media frame into one or more RTP packets and adds the SFrame header to the beginning of the first packet and an auth tag to the end of the last packet.


Encoder
SFrame Enc
Packetizer
DTLS+SRTP
Alice
Key Manager
Encrypted Packe
E2EE channel via the Messaging Server
Messaging Server
Media Server



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

4.1. SFrame Format

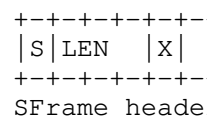


4.2. SFrame Header

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:



SFrame header metadata

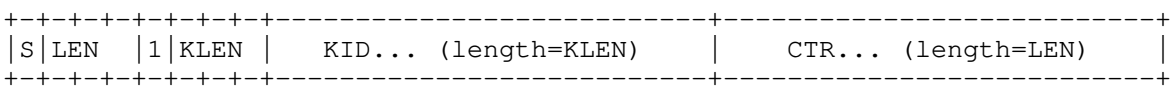
Signature flag (S): 1 bit This field indicates the payload contains a signature if set. Counter Length (LEN): 3 bits This field indicates the length of the CTR fields in bytes. Extended Key Id Flag (X): 1

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:

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:



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

4.3.1. Key Derivation

Each client creates a 32 bytes secret key K and share it with with other participants via an E2EE channel. From K, we derive 3 secrets:

1- Salt key used to calculate the IV

Key = HKDF(K, 'SFrameSaltKey', 16)

2- Encryption key to encrypt the media frame

Key = HKDF(K, 'SFrameEncryptionKey', 16)

3- Authentication key to authenticate the encrypted frame and the media metadata

Key = HKDF(K, 'SFrameAuthenticationKey', 32)

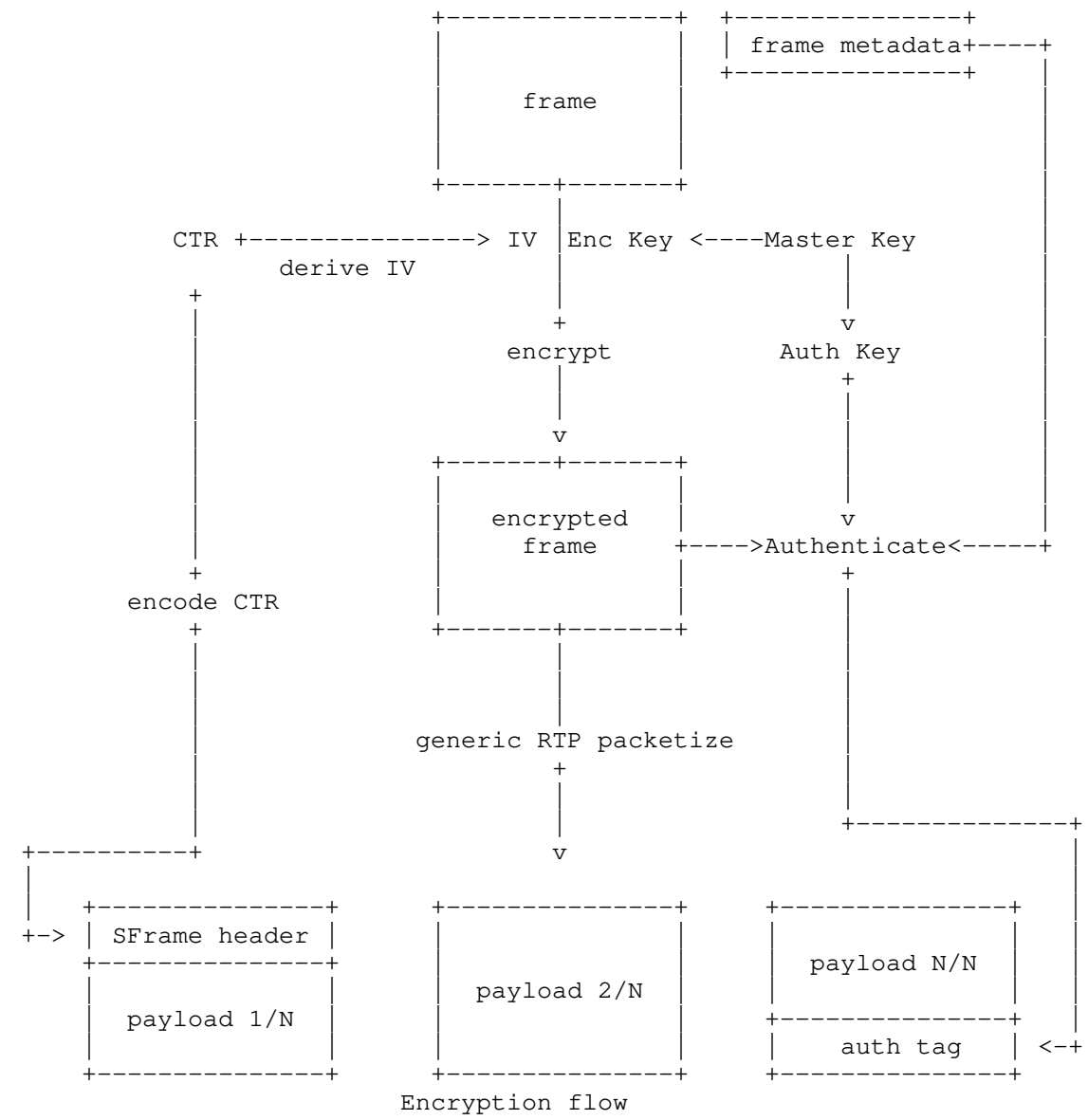
The IV is 128 bits long and calculated from the CTR field of the Frame header:

IV = CTR XOR Salt key

4.3.2. 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. The encryptor constructs SFrame header using frame counter and key id and derive the encryption IV. The frame is encrypted using the encryption key and the header, encrypted frame, the media metadata and the header are authenticated using the authentication key. The authentication tag is then truncated (If supported by the cipher suite) and prepended at the end of the ciphertext.

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



Encryption flow

4.3.3. 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 Frame for decryption. SFrame maintains multiple decryptor objects, one for each client in the call. Initially the client might

not have the mapping between the incoming streams the user's keys, in this case SFrame tries all unmapped keys until it finds one that passes the authentication verification and use it to decrypt the frame. If the client has the mapping ready, it can push it down to SFrame later.

The KeyId field in the SFrame header is used to find the right key for that user, which is incremented by the sender when they switch to a new key.

For frames that are failed to decrypt because there is not key available yet, SFrame will buffer them and retries to decrypt them once a key is received.

4.3.4. 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.

4.3.5. Key Rotation

Because the E2EE keys could be rotated during the call when people join and leave, these new keys are exchanged using the same E2EE secure channel used in the initial key negotiation. Sending new fresh keys is an expensive operation, so the key management component might chose to send new keys only when other clients leave the call and use hash ratcheting for the join case, so no need to send a new key to the clients who are already on the call. SFrame supports both modes

4.3.5.1. Key Ratcheting

When SFrame decryptor fails to decrypt one of the frames, it automatically ratchets the key forward and retries again until one ratchet succeed or it reaches the maximum allowed ratcheting window. If a new ratchet passed the decryption, all previous ratchets are deleted.

K(i) = HKDF(K(i-1), 'SFrameRatchetKey', 32)

4.3.5.2. New Key

SFrame will set the key immediately on the decrypts when it is received and destroys the old key material, so if the key manager sends a new key during the call, it is recommended not to start using it immediately and wait for a short time to make sure it is delivered to all other clients before using it to decrease the number of decryption failure. It is up to the application and the key manager to define how long this period is.

4.4. Authentication

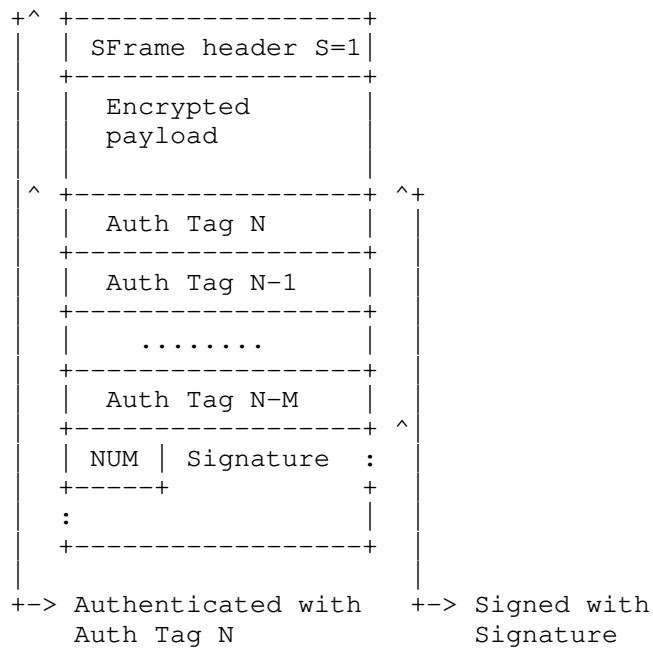
Every client in the call knows the secret key for all other clients so it can decrypt their traffic, it also means a malicious client can impersonate any other client in the call by using the victim key to encrypt their traffic. This might not be a problem for consumer application where the number of 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.



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

4.5.1. SFrame

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

o A hash function This is used for the Key derivation and frame hashes for signature. We recommend using SHA256 hash function.

o An AEAD encryption algorithm [RFC5116] While any AEAD algorithm can be used to encrypt the frame, we recommend using algorithms with safe MAC truncation like AES-CTR and HMAC to reduce the per-frame overhead. In this case we can use 80 bits MAC for video frames and 32 bits for audio frames similar to DTLS-SRTP cipher suites:

1- AES_CM_128_HMAC_SHA256_80

2- AES_CM_128_HMAC_SHA256_32

o [Optional] A signature algorithm If signature is supported, we recommend using ed25519

4.5.2. DTLS-SRTP

SRTP is used as an HBH encryption, since the media payload is already encrypted, and SRTP only protects the RTP headers, one implementation could use 4 bytes outer auth tag to decrease the overhead, however it is up to the application to use other ciphers like AES-128-GCM with full authentication tag.

5. Key Management

SFrame must be integrated with an E2EE key management framework to exchange and rotate the encryption keys. This framework will maintain a group of participant endpoints who are in the call. At call setup time, each endpoint will create a fresh key material and optionally signing key pair for that call and encrypt the key material and the public signing key to every other endpoints. They encrypted keys are delivered by the messaging delivery server using a reliable channel.

The KMS will monitor the group changes, and exchange new keys when necessary. It is up to the application to define this group, for example one application could have ephemeral group for every call and keep rotating key when end points joins or leave the call, while another application could have a persisted group that can be used for multiple calls and exchange keys with all group endpoints for every call.

When a new key material is created during the call, we recommend not to start using it immediately in SFrame to give time for the new keys to be delivered. If the application supports delivery receipts, it can be used to track if the key is delivered to all other endpoints on the call before using it.

Keys must have a sequential id starting from 0 and incremented eery time a new key is generated for this endpoint. The key id will be added in the SFrame header during encryption, so the recipient know which key to use for the decryption.

5.1. MLS-SFrame

While any other E2EE KMS can be used with SFrame, there is a big advantage if it is used with [MLSARCH] which natively supports very large groups efficiently. When [MLSPROTO] is used, the endpoints keys (AKA Application secret) can be used directly for SFrame without the need to exchange separate key material. The application secret is rotated automatically by [MLSPROTO] when group membership changes.

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 bps@20ms	Overhead bps@40ms	Overhead 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 bps@30fps	Overhead bps@15fps	Overhead 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

[PERC] 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. [PERCLITE] <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 [PERCLITE] 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 [PERCLITE] implementation

7.3.1. Audio

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

7.3.2. Video

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

[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

[MLSARCH] Omara, E., Barnes, R., Rescorla, E., Inguva, S., Kwon, A., and A. Duric, "Messaging Layer Security Architecture", 2020.

[MLSPROTO] Barnes, R., Millican, J., Omara, E., Cohn-Gordon, K., and R. Robert, "Messaging Layer Security Protocol", 2020.

[PERC] Jennings, C., Jones, P., Barnes, R., and A. Roach, "PERC", 2020, <https://datatracker.ietf.org/doc/rfc8723/>.

[PERCLITE] GOUAILLARD, A. and S. Murillo, "PERC-Lite", 2020, <https://tools.ietf.org/html/draft-murillo-perc-lite-01>.

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

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 30 September 2021

E. Omara
J. Uberti
Google
A. GOUAILLARD
S. Murillo
CoSMo Software

29 March 2021

Secure Frame (SFrame)
draft-omara-sframe-02

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 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 30 September 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.

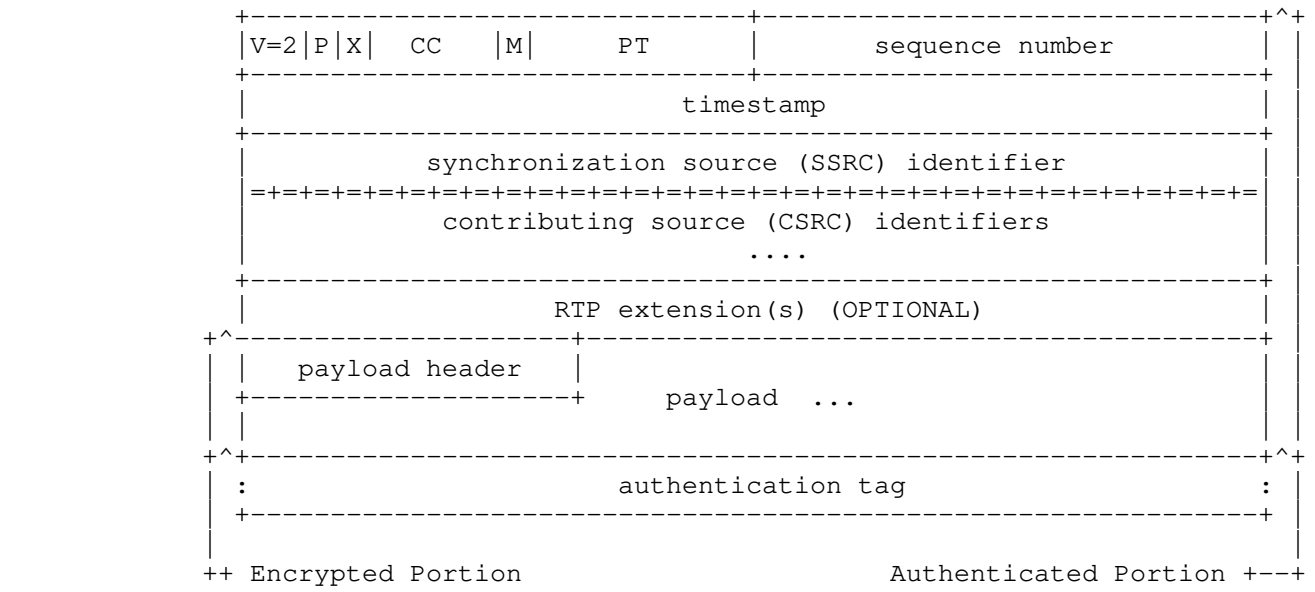


Figure 1: SRTP packet format

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.

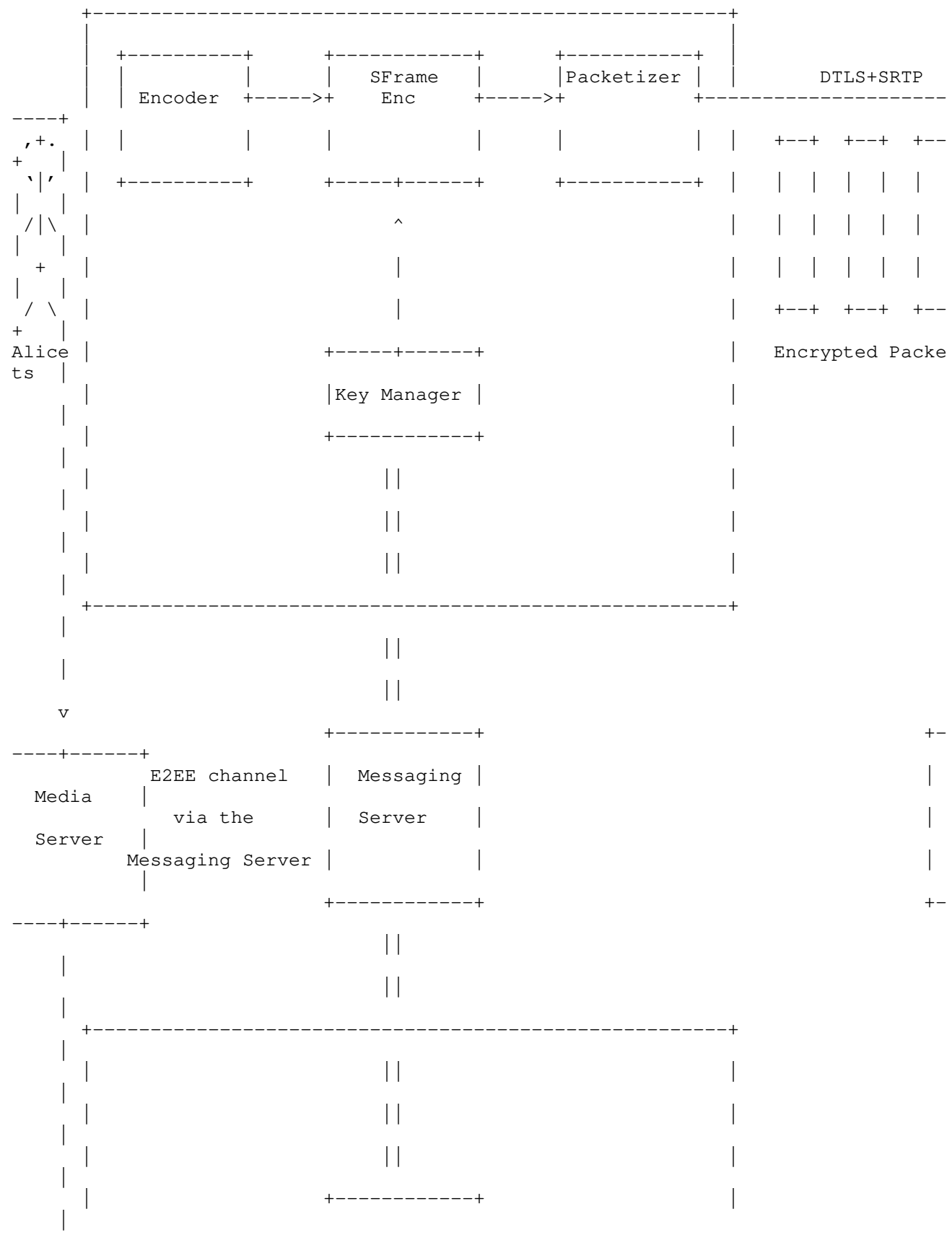
8. Work with the most popular audio and video codecs used in conferencing scenarios.

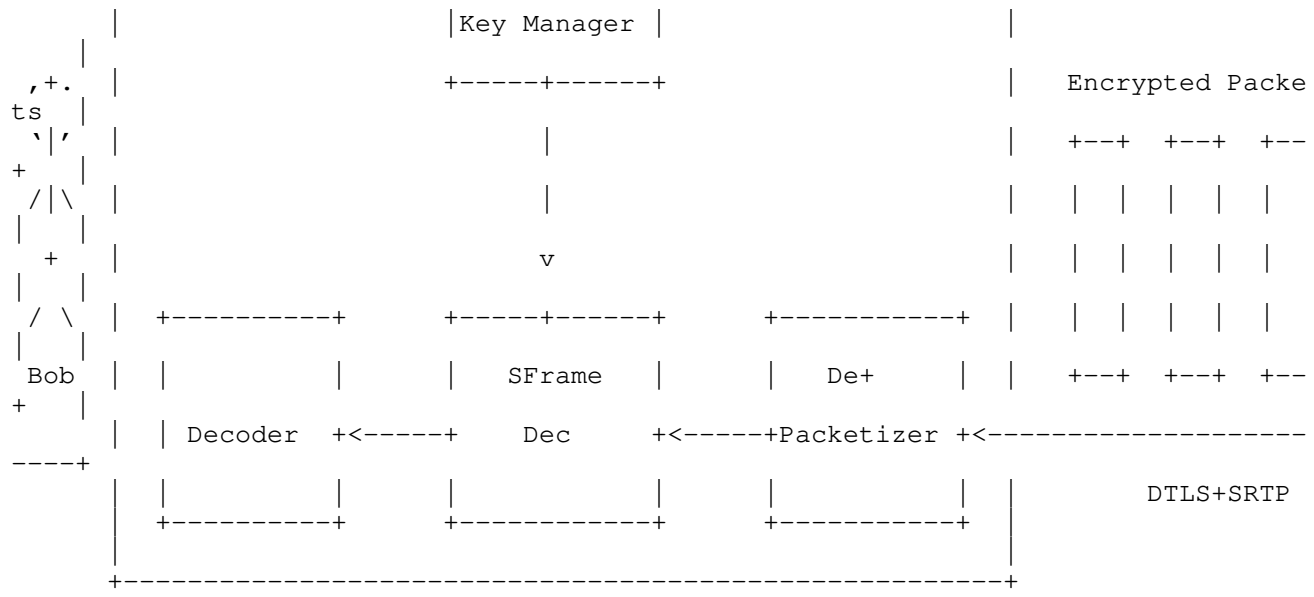
4. SFrame

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.

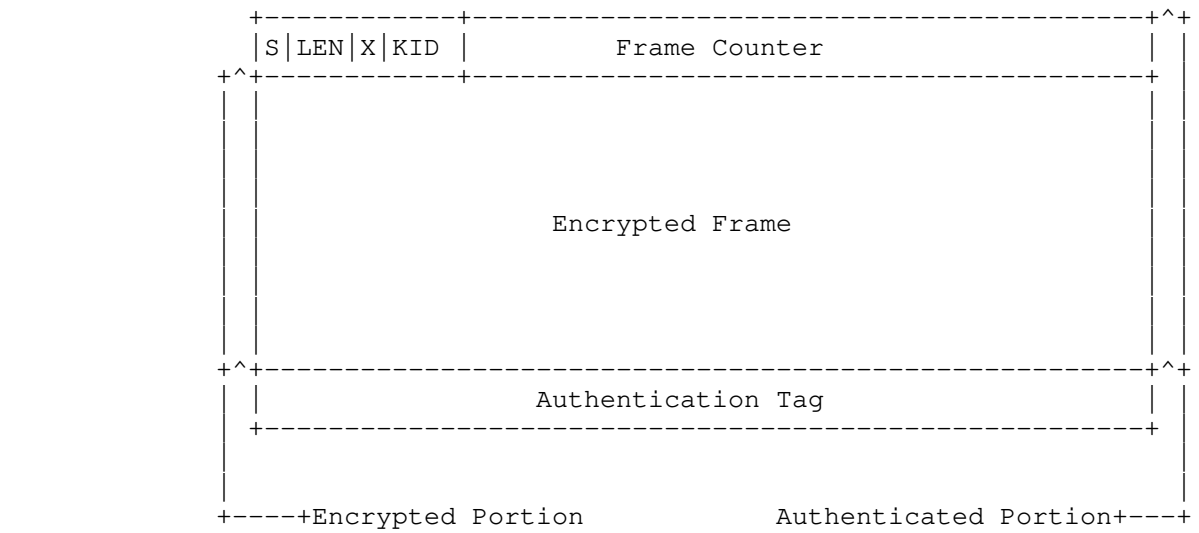
The generic packetizer splits the E2E encrypted media frame into one or more RTP packets and adds the SFrame header to the beginning of the first packet and an auth tag to the end of the last packet.


Encoder
SFrame Enc
Packetizer
DTLS+SRTP
Alice
Key Manager
Encrypted Packe
E2EE channel via the Messaging Server
Messaging Server
Media Server



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

4.1. SFrame Format

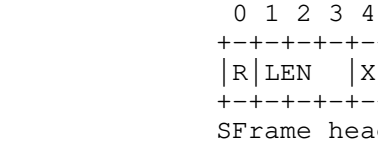


4.2. SFrame Header

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.

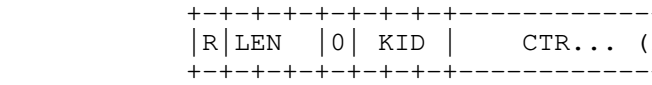
Both the frame counter and the key id are encoded in a variable length format to decrease the overhead, so the first byte in the Sframe header is fixed and contains the header metadata with the following format:



SFrame header metadata

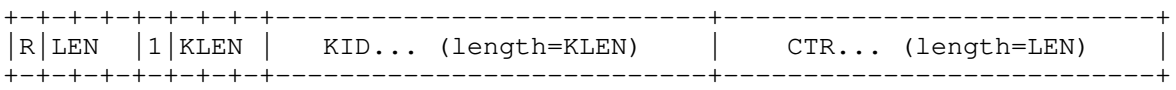
Reserved (R): 1 bit This field MUST be set to zero on sending, and MUST be ignored by receivers. Counter Length (LEN): 3 bits This field indicates the length of the CTR fields in bytes. Extended Key Id Flag (X): 1 bit Indicates if the key field contains the key id or the key length. Key or Key Length: 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:



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:



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 4.4). 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.

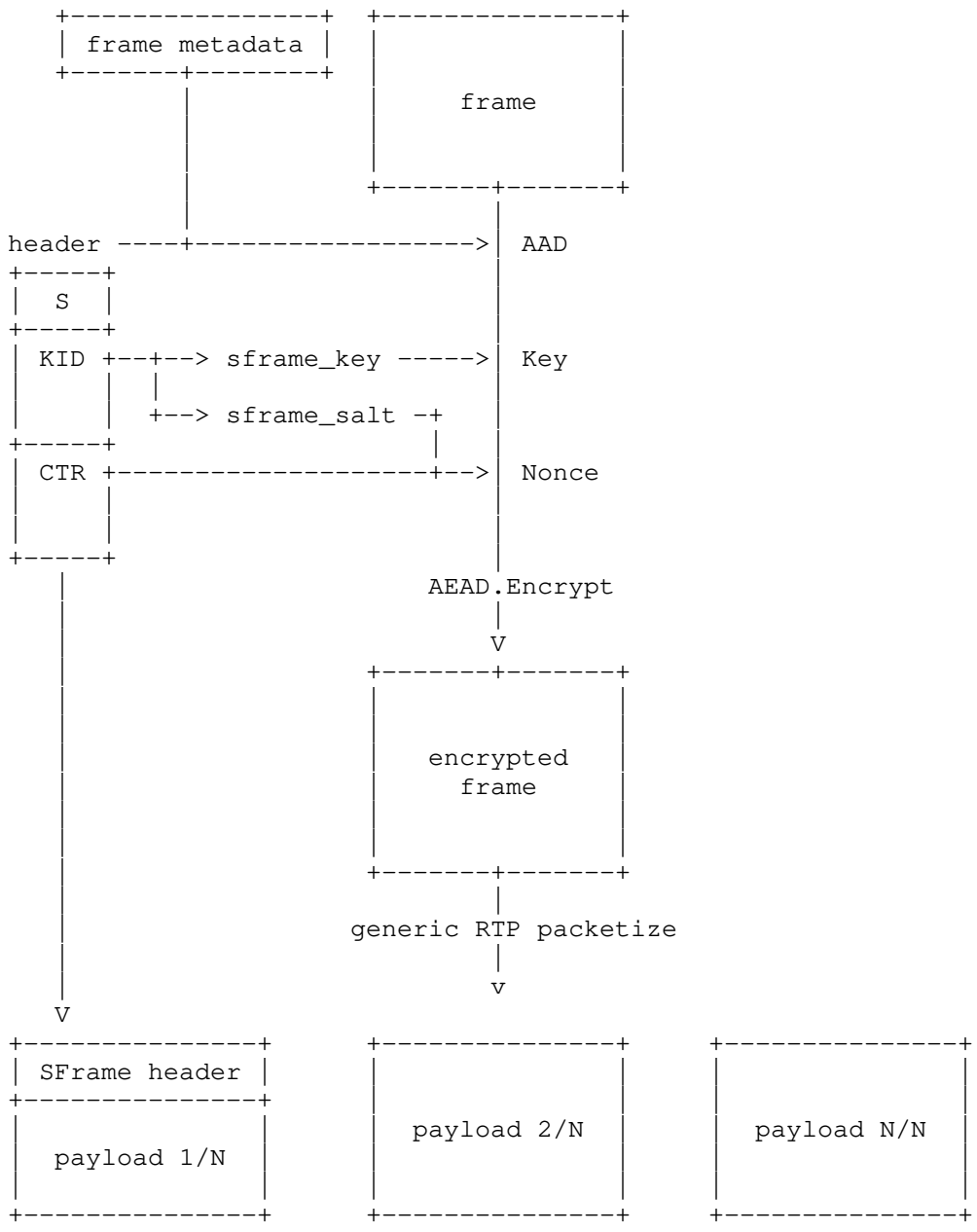


Figure 2: Encryption flow

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. 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

Table 1

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.

4.4.1. 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(nonce, aad, ct):
  aad_len = encode_big_endian(len(aad), 8)
  ct_len = encode_big_endian(len(ct), 8)
  auth_data = aad_len + ct_len + nonce + 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(nonce, aad, ct)
  return ct + tag

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

  candidate_tag = compute_tag(nonce, 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.

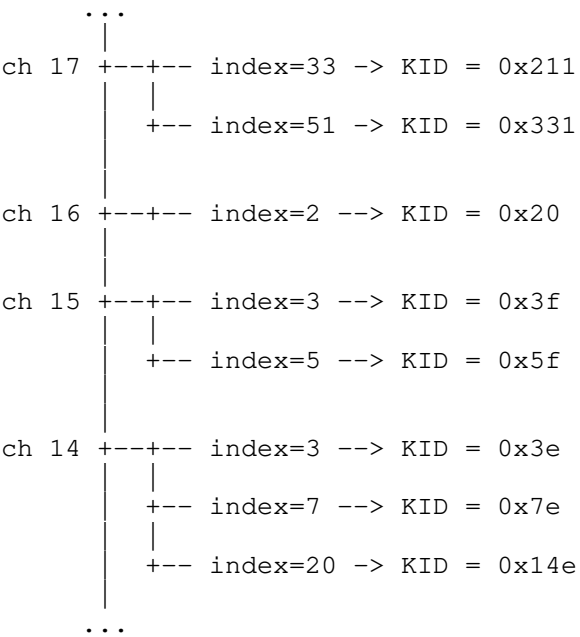
```
sframe_epoch_secret = MLS-Exporter("SFrame 10 MLS", "", AEAD.Nk)

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.



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

Table 2

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 bps@30fps	Overhead bps@15fps	Overhead 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

Table 3

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

Table 4

7.3.2. Video

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

Table 5

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. No Per-Sender Authentication

SFrame does not provide per-sender authentication of media data. Any sender in a session can send media that will be associated with any other sender. This is because SFrame uses symmetric encryption to protect media data, so that any receiver also has the keys required to encrypt packets for the sender.

8.2. 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".

8.3. 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", Work in Progress, Internet-Draft, draft-ietf-mls-architecture-05, 26 July 2020, <http://www.ietf.org/internet-drafts/draft-ietf-mls-architecture-05.txt>.

[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", Work in Progress, Internet-Draft, draft-ietf-mls-protocol-11, 22 December 2020, <http://www.ietf.org/internet-drafts/draft-ietf-mls-protocol-11.txt>.

[I-D.murillo-perc-lite]
Murillo, S. and A. Gouaillard, "End to End Media Encryption Procedures", Work in Progress, Internet-Draft, draft-murillo-perc-lite-01, 12 May 2020, <http://www.ietf.org/internet-drafts/draft-murillo-perc-lite-01.txt>.

[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.B. 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