

Network Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: 12 August 2026

C. Jennings  
S. Nandakumar  
R. Barnes  
Cisco  
8 February 2026

End-to-End Secure Objects for Media over QUIC Transport  
draft-jennings-moq-secure-objects-04

## Abstract

This document specifies an end-to-end authenticated encryption scheme for application objects transmitted via Media over QUIC (MoQ) Transport. The scheme enables original publishers to securely share a symmetric key with end subscribers, ensuring that MoQ relays are unable to decrypt object contents. Additionally, subscribers can verify the integrity and authenticity of received objects, confirming that the content has not been modified in transit.

## About This Document

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

The latest revision of this draft can be found at <https://suhashere.github.io/moq-secure-objects/#go.draft-jennings-moq-secure-objects.html>. Status information for this document may be found at <https://datatracker.ietf.org/doc/draft-jennings-moq-secure-objects/>.

Discussion of this document takes place on the Media over QUIC Working Group mailing list (<mailto:moq@ietf.org>), which is archived at <https://mailarchive.ietf.org/arch/browse/moq/>. Subscribe at <https://www.ietf.org/mailman/listinfo/moq/>.

Source for this draft and an issue tracker can be found at <https://github.com/suhasHere/moq-secure-objects>.

## 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 12 August 2026.

## Copyright Notice

Copyright (c) 2026 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 Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

## Table of Contents

1.	Introduction
1.1.	Terminology
1.2.	Notational Conventions
1.2.1.	Serialized Full Track Name
2.	MoQT Object Model Recap
3.	Secure Objects
3.1.	Extensions
3.2.	Setup Assumptions
3.3.	Application Procedure
3.3.1.	Object Encryption
3.3.2.	Object Decryption
3.4.	Encryption Schema
3.5.	Metadata Authentication
3.6.	Nonce Formation
3.7.	Key Derivation
3.8.	Encryption
3.9.	Decryption
4.	Header Extensions
4.1.	Key ID Extension
4.2.	Private Extension
5.	Usage Considerations
6.	Security Considerations
6.1.	AEAD Invocation Limits
6.2.	Detecting Deletion by Malicious Relays
6.2.1.	Monotonically Incrementing Identifiers
6.2.2.	Non-Sequential Identifiers with Gap Extensions
6.3.	Signaling End of Content
7.	IANA Considerations
7.1.	MOQ Extension Headers Registry
7.2.	Cipher Suites
8.	References
8.1.	Normative References
8.2.	Informative References
Appendix A. Acknowledgements	
Authors' Addresses	

## 1. Introduction

Media Over QUIC Transport (MoQT) is a protocol that is optimized for the QUIC protocol, either directly or via WebTransport, for the dissemination of delivery of low latency media [MoQ-TRANSPORT]. MoQT defines a publish/subscribe media delivery layer across set of participating relays for supporting wide range of use-cases with different resiliency and latency (live, interactive) needs without compromising the scalability and cost effectiveness associated with content delivery networks. It supports sending media objects through sets of relays nodes.

Typically a MOQ Relay doesn't need to access the media content, thus allowing the media to be "end-to-end" encrypted so that it cannot be decrypted by the relays. However for a relay to participate effectively in the media delivery, it needs to access naming information of a MoQT object to carryout the required store and forward functions.

As such, two layers of security are required:

1. Hop-by-hop (HBH) security between two MoQT endpoints.
2. End-to-end (E2E) security from the Original Publisher of an MoQT object to End Subscribers

The HBH security is provided by TLS in the QUIC connection that MoQT runs over. MoQT support different E2EE protection as well as allowing for E2EE security.

This document defines a scheme for E2E authenticated encryption of MoQT objects. This scheme is based on the SFrame mechanism for authenticated encryption of media objects [SFRAME].

However, a secondary goal of this design is to minimize the amount of additional data the encryptions requires for each object. This is particularly important for very low bit rate audio applications where the encryption overhead can increase overall bandwidth usage by a significant percentage. To minimize the overhead added by end-to-end encryption, certain fields that would be redundant between MoQT and SFrame are not transmitted.

The encryption mechanism defined in this specification can only be used in application context where object ID values are never more than 32 bits long.

### 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.

E2EE: End to End Encryption

HBH: Hop By Hop

varint: [MoQ-TRANSPORT] variable length integer. TODO - align with -17 MoQT draft

### 1.2. Notational Conventions

#### 1.2.1. Serialized Full Track Name

Serialized Full Track Name is composed of MoQT Track Namespace and Track Name as shown below:

Serialized Full Track Name = Serialize(Track Namespace) + Serialize(Track Name)

The Serialize operation follows the same on-the-wire encoding for Track Name Space and Track Name as defined in Section 2.4.1 of [MoQ-TRANSPORT].

This mandates that the serialization of Track Namespace tuples starts with varint encoded count of tuples. This is followed by encoding corresponding to each tuple. Each tuple's encoding starts with varint encoded length for the count of bytes and bytes representing the content of the tuple.

The Track Name is varint encoded length followed by sequence of bytes that identifies an individual track within the namespace.

The + represents concatenation of byte sequences.

## 2. MoQT Object Model Recap

MoQT defines a publish/subscribe based media delivery protocol, where in endpoints, called original publishers, publish objects which are delivered via participating relays to receiving endpoints, called end subscribers.

Section 2 of [MoQ-TRANSPORT] defines hierarchical object model for application data, comprised of objects, groups and tracks.

Objects defines the basic data element, an addressable unit whose payload is sequence of bytes. All objects belong to a group, indicating ordering and potential dependencies. A track contains has collection of groups and serves as the entity against which a subscribers issue subscription requests.

Media Over QUIC Application

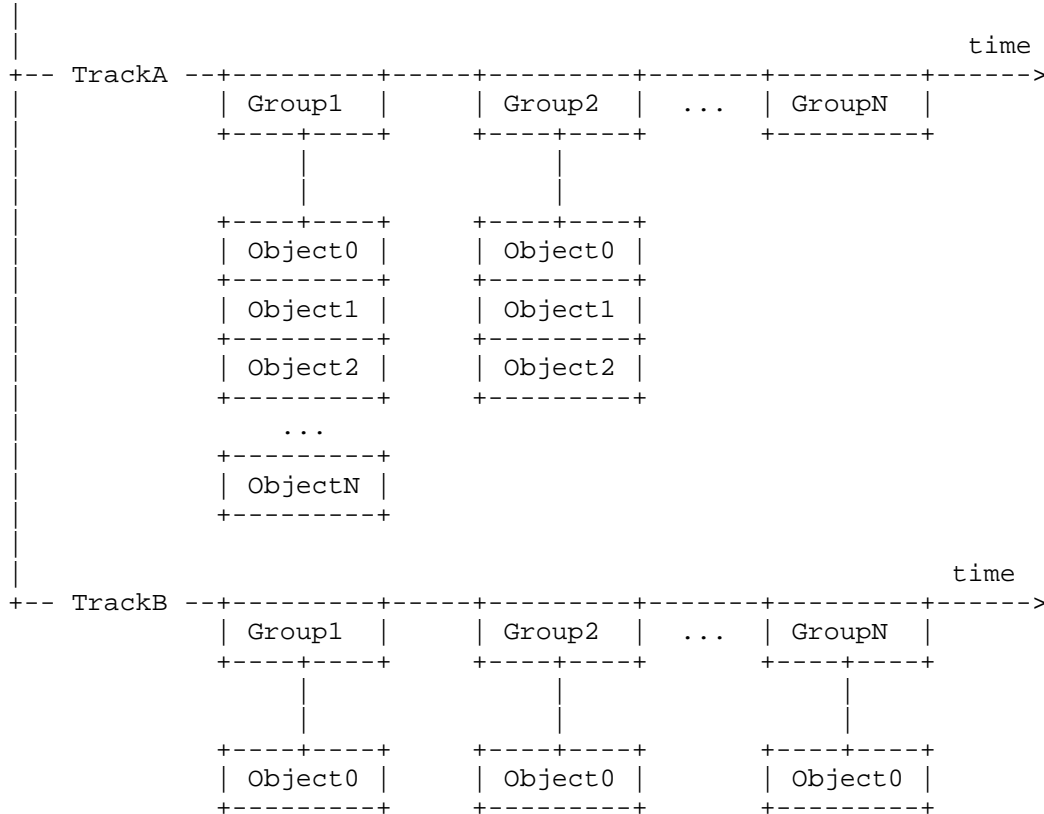


Figure 1: Structure of an MoQT session

Objects are comprised of three parts: parts that Relays can read and modify, parts that Relay can read but is not allowed to modify, and parts the Relays cannot read or modify. The payload portion MAY be end to end encrypted, in which case it is only visible to the original publisher and the end subscribers. The application is solely responsible for the content of the object payload.

Tracks are identified by a combination of its Track Namespace and Track Name. Tuples of the Track Namespace and Track Name are treated as a sequence of binary bytes. Group and Objects are represented as variable length integers called GroupID and ObjectID respectively.

Two important properties of objects are:

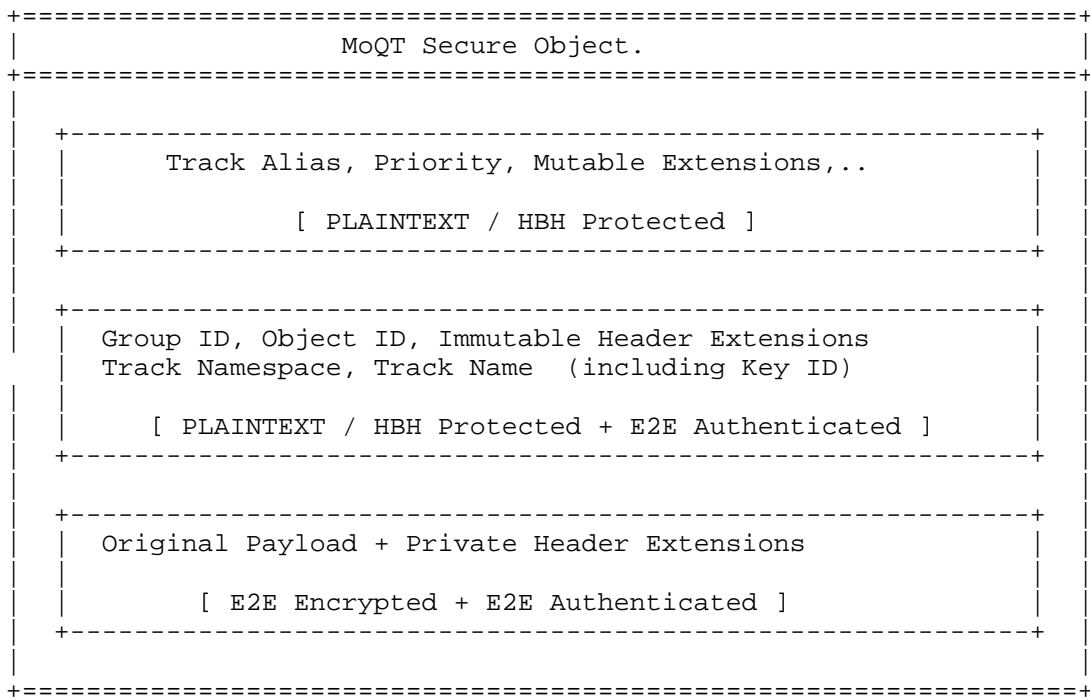
1. The combination of Track Namespace, Track Name, Group ID and Object ID are globally unique in a given relay network, referred to as Full Object ID in this specification.
2. The data inside an MoQT Object (and its size) can never change after the Object is first published. There can never be two

Objects with the same Full Object ID but different data.

One of the ways system keep the Full Object IDs unique is by using a fully qualified domain names or UUIDs as part of the Track Namespace.

### 3. Secure Objects

Section 10.2.1 [MoQ-TRANSPORT] defines fields of a canonical MoQT Object. The protection scheme defined in this draft encrypts the Object Payload and Private header extensions Section 4.2. The scheme authenticates the Group ID, Object ID, Immutable Header Extensions (Section 11.2 of [MoQ-TRANSPORT] }} and Object Payload fields, regardless of the on-the-wire encoding of the objects over QUIC Datagrams or QUIC streams.



#### Legend:

PLAINTEXT / HBH Protected: Visible to relays, protected by TLS

E2E Authenticated: Integrity protected from original publisher to end subscriber

E2E Encrypted: Confidentiality protected from original publisher to end subscriber

Figure 2: MoQ Object Structure and Security Protection

#### 3.1. Extensions

MoQT defines two types of Object Header Extensions, mutable and immutable. This specification uses MoQT immutable extensions to convey end-to-end authenticated metadata and adds Private Object header extensions (see Section 4.2). Private extensions are serialized and encrypted along with the Object payload, decrypted and deserialized by the receiver. This specification further defines Secure Object KID extension (see Section 4.1), which is transmitted within the immutable extensions.

#### 3.2. Setup Assumptions

The application assigns each track a set of (Key ID, track\_base\_key) tuples, where each track\_base\_key is known only to authorized original publishers and end subscribers for a given track. How these per-track secrets and their lifetimes are established is outside the scope of this specification. The application also defines which Key ID should be used for a given encryption operation. For decryption,

the Key ID is obtained from the Secure Object KID header extension (that is contained within the immutable header extension of the Object).

Applications determine the ciphersuite to be used for each track's encryption context. See Section 7.2 for the list of ciphersuites that can be used.

### 3.3. Application Procedure

This section provides steps for applications over MoQT to use mechanisms defined in this specification.

#### 3.3.1. Object Encryption

To encrypt a MoQT Object, the application constructs a plaintext from the application data and any private header extensions:

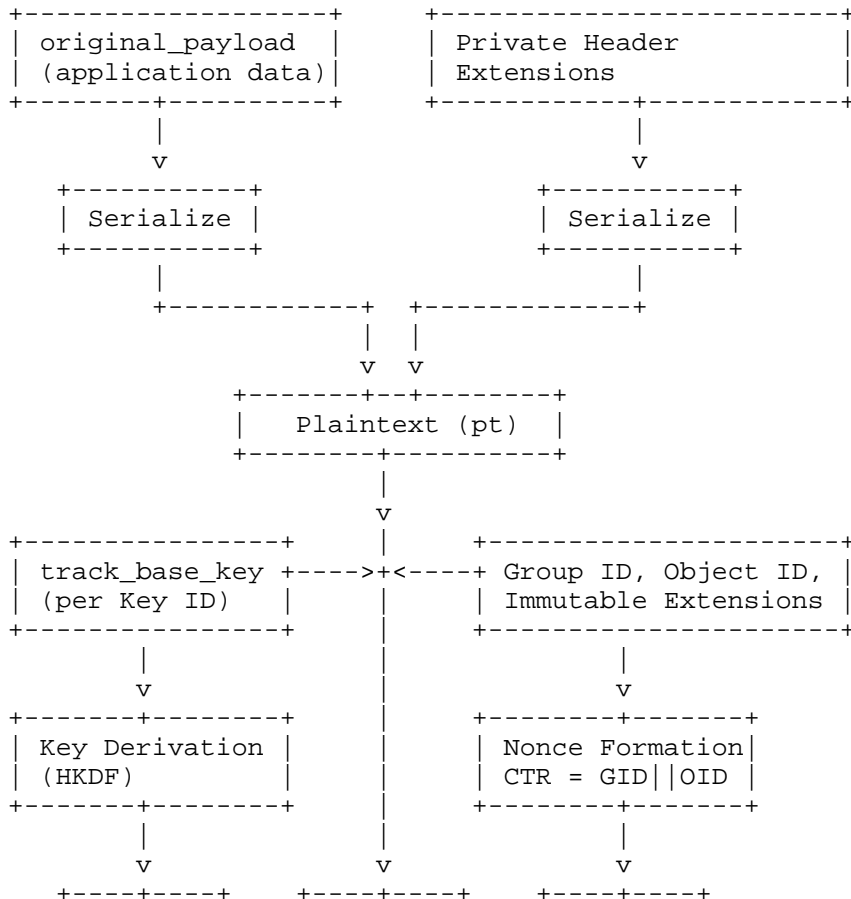
```
pt = Serialize(original_payload) + Serialize(Private header extensions)
```

Where `original_payload` is the application's object data. The serialization of `original_payload` consists of a varint-encoded byte count followed by the payload bytes. The serialization for private header extensions follows the rules for immutable extensions (as defined in section 11 of [MoQ-TRANSPORT]).

The plaintext is then encrypted:

```
ciphertext = encrypt(pt)
```

The resulting ciphertext replaces the `original_payload` as the MoQT Object Payload. The ciphertext length reflects the encrypted `original_payload` plus any private header extensions plus the AEAD authentication tag.



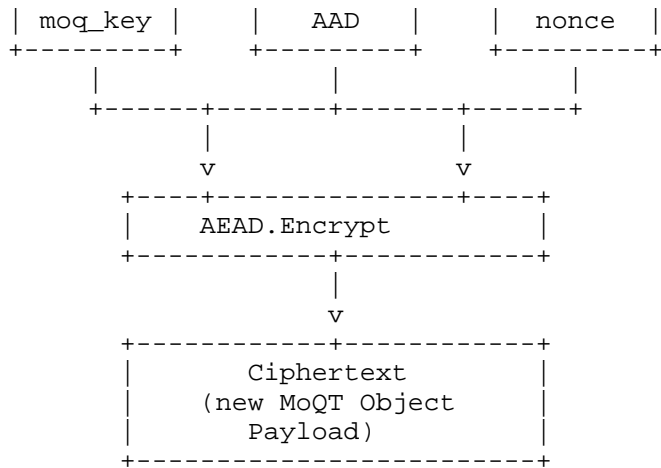


Figure 3: Object Encryption Process

### 3.3.2. Object Decryption

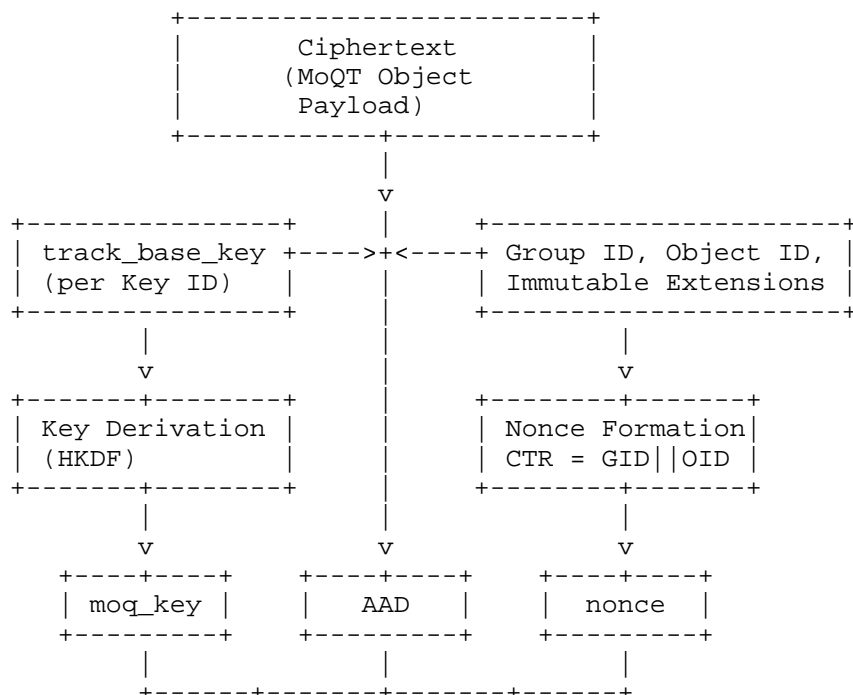
To decrypt a MoQT Object, the application provides the MoQT Object Payload as ciphertext input to obtain the plaintext:

```
pt = decrypt(ciphertext)
```

The plaintext is then deserialized to extract the application's original\_payload and any private header extensions:

1. Read a varint to obtain the original\_payload length.
2. Read that many bytes as original\_payload.
3. If no bytes remain, there are no private header extensions.
4. Otherwise, read the extension type (16 bits). If the value is not 0xA, drop the object. Parse the remaining bytes as the Private header extension structure.

If parsing fails at any stage, the receiver MUST drop the MoQT Object.



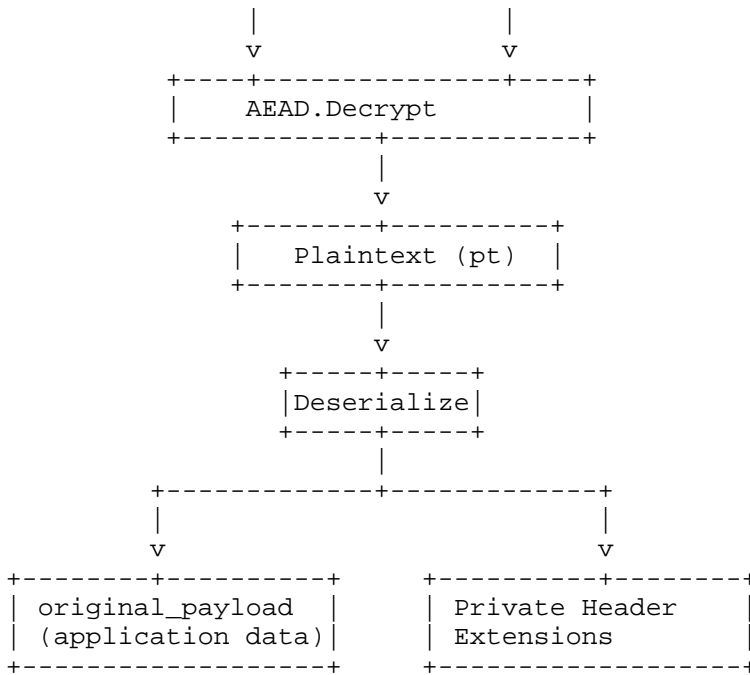


Figure 4: Object Decryption Process

### 3.4. Encryption Schema

MoQT secure object protection relies on an ciphersuite to define the AEAD encryption algorithm and hash algorithm in use (Section 7.2). We will refer to the following aspects of the AEAD and the hash 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 in bytes of a key for the encryption algorithm
- \* AEAD.Nn - The size in bytes of a nonce for the encryption algorithm
- \* AEAD.Nt - The overhead in bytes of the encryption algorithm (typically the size of a "tag" that is added to the plaintext)
- \* AEAD.Nka - For cipher suites using the compound AEAD described in (Section 4.5.1 of [SFRAME]), the size in bytes of a key for the underlying encryption algorithm
- \* Hash.Nh - The size in bytes of the output of the hash function

### 3.5. Metadata Authentication

The Key ID, Full Track Name, Immutable Header Extensions, Group ID, and Object ID for a given MoQT Object are authenticated as part of secure object encryption. This ensures, for example, that encrypted objects cannot be replayed across tracks.

When protecting or unprotecting a secure object, the following data structure captures the input to the AEAD function's AAD argument:

```

SECURE_OBJECT_AAD {
    Key ID (i),
    Group ID (i),

```

```

    Object ID (i),
    Track Namespace (...),
    Track Name Length (i),
    Track Name (...),
    Serialized Immutable Extensions (...)
}

```

\* Track Namespace is serialized as in section 2.4.1 of MoQT.

Serialized Immutable Extensions MUST include the Secure Object KID header extension containing the Key ID.

### 3.6. Nonce Formation

The Group ID and Object ID for an object are used to form a 96-bit counter (CTR) value, which XORed with a salt to form the nonce used in AEAD encryption. The counter value is formed by bitwise concatenating the Group ID as 64 bit integer and Object ID as 32 bit integer. This encryption/decryption will fail if applied to an object where group ID is larger than  $2^{64}$  or the object ID is larger than  $2^{32}$  and the MoQT Object MUST NOT be processed further.

### 3.7. Key Derivation

Encryption and decryption use a key and salt derived from the `track_base_key` associated with a Key ID. Given a `track_base_key` value, the key and salt are derived using HMAC-based Key Derivation Function (HKDF) [RFC5869] as follows:

```

def derive_key_salt(key_id, track_base_key, serialized_full_track_name):
    moq_secret = HKDF-Extract("", track_base_key)
    moq_key_label = "MOQ 1.0 Secure Objects Secret key " + serialized_full_track_name +
cipher_suite + key_id
    moq_key =
        HKDF-Expand(moq_secret, moq_key_label, AEAD.Nk)
    moq_salt_label = "MOQ 1.0 Secret salt " + serialized_full_track_name + cipher_suite
+ key_id
    moq_salt =
        HKDF-Expand(moq_secret, moq_salt_label, AEAD.Nn)

    return moq_key, moq_salt

```

In the derivation of `moq_secret`:

- \* The + operator represents concatenation of byte strings.
- \* The Key ID value is encoded as an 8-byte big-endian integer.
- \* The `cipher_suite` value is a 2-byte big-endian integer representing the cipher suite in use (see [SFRAME]).

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

### 3.8. Encryption

MoQT secure object encryption uses the AEAD encryption algorithm for the cipher suite in use. The key for the encryption is the `moq_key` derived from the `track_base_key` Section 3.7. The nonce is formed by first XORing the `moq_salt` with the current CTR value Section 3.6, and then encoding the result as a big-endian integer of length `AEAD.Nn`.

The Private extensions and Object payload field from the MoQT object is used by the AEAD algorithm for the plaintext.

The encryptor forms an `SecObj` header using the Key ID value provided.

The encryption procedure is as follows:

1. Obtain the plaintext payload to encrypt from the MoQT object. Extract the Group ID, Object ID, and the Serialized Immutable Header Extension from the MoQT object headers. Ensure the Secure Object KID header extension is included, with the Key ID set as its value.
2. Retrieve the `moq_key` and `moq_salt` matching the Key ID.
3. Form the aad input as described in Section 3.5.
4. Form the nonce by as described in Section 3.6.
5. Apply the AEAD encryption function with `moq_key`, nonce, aad, MoQT Object payload and serialized private header extensions as inputs (see Section 3.3).

The final SecureObject is formed from the MoQT transport headers, followed by the output of the encryption.

### 3.9. Decryption

For decrypting, the Key ID from the Secure Object KID header extension contained within immutable header extension is used to find the right key and salt for the encrypted object. The MoQT track information matching the Key ID along with Group ID and Object ID fields of the MoQT object header are used to form the nonce.

The decryption procedure is as follows:

1. Parse the SecureObject to obtain Key ID, the ciphertext corresponding to MoQT object payload and the Group ID and Object ID from the MoQT object headers.
2. Retrieve the `moq_key`, `moq_salt` and MoQT track information matching the Key ID.
3. Form the aad input as described in Section 3.5.
4. Form the nonce by as described in Section 3.6.
5. Apply the AEAD decryption function with `moq_key`, nonce, aad and ciphertext as inputs.
6. Decode the private extension headers, returning both the headers and the object payload.

If extracting Key ID fails either due to missing Secure Object KID extension within immutable header extension or error from parsing, the client MUST discard the received MoQT Object.

If a ciphertext fails to decrypt because there is no key available for the Key ID value presented, the client MAY buffer the ciphertext and retry decryption once a key with that Key ID is received. If a ciphertext fails to decrypt for any other reason, the client MUST discard the ciphertext. Invalid ciphertexts SHOULD be discarded in a way that is indistinguishable (to an external observer) from having processed a valid ciphertext. In other words, the decryption operation should take the same amount of time regardless of whether decryption succeeds or fails.

## 4. Header Extensions

### 4.1. Key ID Extension

Key ID (Extension Header Type 0x2) is a variable length integer and identifies the keying material (keys, nonces and associated context for the MoQT Track) to be used for a given MoQT track.

The Key ID extension is included within the Immutable Header extension. All objects encoded MUST include the Key ID header extension when using this specification for object encryption.

## 4.2. Private Extension

The Private Extensions (Extension Header Type 0xA) contains a sequence of Key-Value-Pairs (see section 1.4.2 [MoQ-TRANSPORT]) which are also Object Extension Headers of the Object. This extension can be added by the Original Publisher and considered part of the Object Payload.

```
Private Extensions {  
    Type (0xA),  
    Length (i),  
    Key-Value-Pair (...) ...  
}
```

## 5. Usage Considerations

To implement the protection mechanisms specified herein, a secure object requires the complete object before any validity checks can be performed. This introduces latency proportional to the object size; if the application aggregates excessive data into a single object (e.g., encapsulating 6 seconds of video), the entire segment must be received before processing or validation can commence, delaying access to all contained data until transfer completion.

## 6. Security Considerations

The cryptographic computations described in this document are exactly those performed in the SFrame encryption scheme defined in [SFRAME]. The scheme in this document is effectively a "virtualized" version of SFrame:

- \* The CTR value used in nonce formation is not carried in the object payload, but instead synthesized from the GroupID and ObjectID.
- \* The AAD for the AEAD operation is not sent on the wire (as with the SFrame Header), but constructed locally by the encrypting and decrypting endpoints.
- \* The format of the AAD is different:
  - The SFrame Header is constructed using MoQT-style varints, instead of the variable-length integer scheme defined in SFrame.
  - The GroupID and GroupID are sent directly, not as the packed CTR value.
- \* The metadata input in to SFrame operations is defined to be the FullTrackName value for the object.
- \* The labels used in key derivation reflect MOQ usage, not generic SFrame.

The security considerations discussed in the SFrame specification thus also apply here.

The SFrame specification lists several things that an application

needs to account for in order to use SFrame securely, which are all accounted for here:

1. *\*Header value uniqueness:* Uniqueness of CTR values follows from the uniqueness of MoQT (GroupID, ObjectID) pairs. We only use one Key ID value, but instead use distinct SFrame contexts with distinct keys per track. This assures that the same (track\_base\_key, Key ID, CTR) tuple is never used twice.
2. *\*Key management:* We delegate this to the MoQT application, with subject to the assumptions described in Section 3.2.
3. *\*Anti-replay:* Replay is not possible within the MoQT framework because of the uniqueness constraints on ObjectIDs and objects, and because the group ID and object ID are cryptographically bound to the secure object payload.
4. *\*Metadata:* The analogue of the SFrame metadata input is defined in Section 3.5.

Any of the SFrame ciphersuites defined in the IANA SFrame Cipher Suites registry [CIPHERS] can be used to protect MoQT objects. The caution against short tags in Section 7.5 of [SFRAME] still applies here, but the MoQT environment provides some safeguards that make it safer to use short tags, namely:

- \* MoQT has hop-by-hop protections provided by the underlying QUIC layer, so a brute-force attack could only be mounted by a relay.
- \* In some usecases MoQT tracks have predictable object arrival rates, so a receiver can interpret a large deviation from this rate as a sign of an attack.
- \* The the binding of the secure object payload to other MoQT parameters (as metadata), together with MoQT's uniqueness properties ensure that a valid secure object payload cannot be replayed in a different context.

## 6.1. AEAD Invocation Limits

AEAD algorithms have limits on how many times a single key can be used before the cryptographic guarantees begin to degrade. Exceeding these limits can compromise confidentiality (allowing an attacker to distinguish encrypted content from random data) or integrity (allowing an attacker to forge valid ciphertexts). The severity of these risks depends on the specific algorithm in use.

Implementations MUST track the number of encryption and decryption operations performed with each moq\_key and ensure that these counts remain within the limits specified in [AEAD-LIMITS] for the cipher suite in use. When approaching these limits, implementations MUST arrange for new keying material to be established (e.g., by rotating to a new Key ID with a fresh track\_base\_key) before the limits are exceeded.

For the AES-GCM cipher suites defined in this document, the primary concern is the confidentiality limit, which restricts the number of encryptions performed with a single key. For AES-CTR-HMAC cipher suites, both encryption and decryption operations count toward the applicable limits.

## 6.2. Detecting Deletion by Malicious Relays

A malicious relay could selectively delete objects or groups before forwarding them to subscribers. While this specification does not mandate detection of such deletions, it does provide mechanisms that

applications can use to detect when content has been removed.

Some applications may not require deletion detection, or may be able to detect missing data based on the internal structure of the object payload (e.g., sequence numbers embedded in the media format). For applications that do require deletion detection at the MoQT layer, the following approaches are available:

#### 6.2.1. Monotonically Incrementing Identifiers

Applications that assign Group IDs and Object IDs in a strictly monotonic sequence (incrementing by 1 for each successive group or object) can straightforwardly detect gaps. A subscriber receiving Group ID N followed by Group ID N+2, or Object ID M followed by Object ID M+3, can conclude that intervening content was not delivered.

#### 6.2.2. Non-Sequential Identifiers with Gap Extensions

Applications that use Group IDs or Object IDs with intentional gaps (e.g., for sparse data or timestamp-based identifiers) MUST include the Group ID Gap and/or Object ID Gap extensions as immutable header extensions. These extensions indicate the expected distance to the next identifier. If the Object ID Gap extension is absent from a secure object, receivers MUST assume a gap value of 1. Similarly, if the Group ID Gap extension is absent, receivers MUST assume a gap value of 1.

#### 6.3. Signaling End of Content

For applications that need to reliably detect lost objects at the end of a subgroup, group, or track, it is RECOMMENDED to signal completion using object status values defined in [MoQ-TRANSPORT]. By explicitly marking the final object in a sequence, subscribers can distinguish between "more objects may arrive" and "all objects have been sent," enabling detection of trailing deletions that would otherwise be undetectable.

### 7. IANA Considerations

#### 7.1. MOQ Extension Headers Registry

This document defines new MoQT Object extension headers under the MOQ Extension Headers registry.

+=====+		
Type	Value	
+=====+		
0x2	Secure Object KID - see Section 4.1	
+-----+		
0xA	Private Extensions - see Section 4.2	
+-----+		

Table 1

#### 7.2. Cipher Suites

This document establishes a "MoQ Secure Objects Cipher Suites" registry. Each cipher suite specifies an AEAD encryption algorithm and a hash algorithm used for key derivation.

The following values are defined for each cipher suite:

- \* Nh: The size in bytes of the hash function output
- \* Nka: The size in bytes of the encryption key for the underlying

cipher (CTR suites only)

- \* Nk: The size in bytes of the AEAD key
- \* Nn: The size in bytes of the AEAD nonce
- \* Nt: The size in bytes of the AEAD authentication tag

Value	Name	R	Nh	Nka	Nk	Nn	Nt
0x0000	Reserved	-					
0x0001	AES_128_CTR_HMAC_SHA256_80	Y	32	16	48	12	10
0x0002	AES_128_CTR_HMAC_SHA256_64	Y	32	16	48	12	8
0x0003	AES_128_CTR_HMAC_SHA256_32	N	32	16	48	12	4
0x0004	AES_128_GCM_SHA256_128	Y	32	n/a	16	12	16
0x0005	AES_256_GCM_SHA512_128	Y	64	n/a	32	12	16
0xF000-0xFFFF	Reserved for private use	-					

Table 2

The "R" column indicates whether the cipher suite is Recommended:

- \* Y: Indicates that the IETF has consensus that the item is RECOMMENDED. Requires Standard Action as defined [RFC8126].
- \* N: Indicates the IETF has made no statement about the suitability of the associated mechanism. Requires First Come First Serve as defined in [RFC8126].
- \* D: Indicates that the item is discouraged and SHOULD NOT be used. Requires Standard Action or IESG Approval as defined in [RFC8126].

Cipher suite values are 2-byte big-endian integers.

\*AES-GCM cipher suites\* (0x0004, 0x0005) use AES-GCM for authenticated encryption with a full 128-bit authentication tag.

\*AES-CTR-HMAC cipher suites\* (0x0001, 0x0002, 0x0003) use AES in counter mode combined with HMAC for authentication in an encrypt-then-MAC construction. These suites support truncated authentication tags, providing lower overhead at the cost of reduced forgery resistance.

Implementations MUST support AES\_128\_GCM\_SHA256\_128 (0x0004).  
Implementations SHOULD support AES\_128\_CTR\_HMAC\_SHA256\_80 (0x0001).

8. References

8.1. Normative References

[AEAD-LIMITS]  
Gundther, F., Thomson, M., and C. A. Wood, "Usage Limits on AEAD Algorithms", Work in Progress, Internet-Draft, draft-irtf-cfrg-aead-limits-11, 4 December 2025, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-aead-limits-11>>.

[MoQ-TRANSPORT]

Nandakumar, S., Vasiliev, V., Swett, I., and A. Frindell,  
"Media over QUIC Transport", Work in Progress, Internet-  
Draft, draft-ietf-moq-transport-16, 13 January 2026,  
<<https://datatracker.ietf.org/doc/html/draft-ietf-moq-transport-16>>.

[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/rfc/rfc2119>>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated  
Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008,  
<<https://www.rfc-editor.org/rfc/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/rfc/rfc5869>>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for  
Writing an IANA Considerations Section in RFCs", BCP 26,  
RFC 8126, DOI 10.17487/RFC8126, June 2017,  
<<https://www.rfc-editor.org/rfc/rfc8126>>.

[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/rfc/rfc8174>>.

[SFRAME] Omara, E., Uberti, J., Murillo, S. G., Barnes, R., Ed.,  
and Y. Fablet, "Secure Frame (SFrame): Lightweight  
Authenticated Encryption for Real-Time Media", RFC 9605,  
DOI 10.17487/RFC9605, August 2024,  
<<https://www.rfc-editor.org/rfc/rfc9605>>.

## 8.2. Informative References

[CIPHERS] IANA, "SFrame Cipher Suites",  
<<https://www.iana.org/assignments/sframe/sframe.xhtml#sframe-cipher-suites>>.

[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/rfc/rfc3711>>.

## Appendix A. Acknowledgements

Thanks to Alan Frindell for providing text on adding private  
extensions. Thank you to Magnus Westerlund for doing a thorough  
security review.

## Authors' Addresses

Cullen Jennings  
Cisco  
Email: [fluffy@cisco.com](mailto:fluffy@cisco.com)

Suhas Nandakumar  
Cisco  
Email: [snandaku@cisco.com](mailto:snandaku@cisco.com)

Richard Barnes

Cisco  
Email: rlb@ipv.sx