Stream: Internet Engineering Task Force (IETF)
RFC: 9001
Category: Standards Track
Published: May 2021
ISSN: 2070-1721
Authors: M. Thomson, Ed. *Mozilla*
S. Turner, Ed. *sn3rd*

RFC 9001
Using TLS to Secure QUIC

Abstract

This document describes how Transport Layer Security (TLS) is used to secure QUIC.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9001.

Copyright Notice

Table of Contents

1. Introduction
2. Notational Conventions
   2.1. TLS Overview
3. Protocol Overview
4. Carrying TLS Messages
   4.1. Interface to TLS
      4.1.1. Handshake Complete
      4.1.2. Handshake Confirmed
      4.1.3. Sending and Receiving Handshake Messages
      4.1.4. Encryption Level Changes
      4.1.5. TLS Interface Summary
   4.2. TLS Version
   4.3. ClientHello Size
   4.4. Peer Authentication
   4.5. Session Resumption
   4.6. 0-RTT
      4.6.1. Enabling 0-RTT
      4.6.2. Accepting and Rejecting 0-RTT
      4.6.3. Validating 0-RTT Configuration
   4.7. HelloRetryRequest
   4.8. TLS Errors
   4.9. Discarding Unused Keys
      4.9.1. Discarding Initial Keys
      4.9.2. Discarding Handshake Keys
      4.9.3. Discarding 0-RTT Keys
5. Packet Protection
   5.1. Packet Protection Keys
   5.2. Initial Secrets

5.3. AEAD Usage
5.4. Header Protection
5.4.1. Header Protection Application
5.4.2. Header Protection Sample
5.4.3. AES-Based Header Protection
5.4.4. ChaCha20-Based Header Protection
5.5. Receiving Protected Packets
5.6. Use of 0-RTT Keys
5.7. Receiving Out-of-Order Protected Packets
5.8. Retry Packet Integrity
6. Key Update
6.1. Initiating a Key Update
6.2. Responding to a Key Update
6.3. Timing of Receive Key Generation
6.4. Sending with Updated Keys
6.5. Receiving with Different Keys
6.6. Limits on AEAD Usage
6.7. Key Update Error Code
7. Security of Initial Messages
8. QUIC-Specific Adjustments to the TLS Handshake
8.1. Protocol Negotiation
8.2. QUIC Transport Parameters Extension
8.3. Removing the EndOfEarlyData Message
8.4. Prohibit TLS Middlebox Compatibility Mode
9. Security Considerations
9.1. Session Linkability
9.2. Replay Attacks with 0-RTT
9.3. Packet Reflection Attack Mitigation
9.4. Header Protection Analysis
9.5. Header Protection Timing Side Channels

9.6. Key Diversity
9.7. Randomness
10. IANA Considerations
11. References
11.1. Normative References
11.2. Informative References
Appendix A. Sample Packet Protection
A.1. Keys
A.2. Client Initial
A.3. Server Initial
A.4. Retry
A.5. ChaCha20-Poly1305 Short Header Packet
Appendix B. AEAD Algorithm Analysis
B.1. Analysis of AEAD_AES_128_GCM and AEAD_AES_256_GCM Usage Limits
B.1.1. Confidentiality Limit
B.1.2. Integrity Limit
B.2. Analysis of AEAD_AES_128_CCM Usage Limits
Contributors
Authors' Addresses

1. Introduction

This document describes how QUIC [QUIC-TRANSPORT] is secured using TLS [TLS13].

TLS 1.3 provides critical latency improvements for connection establishment over previous versions. Absent packet loss, most new connections can be established and secured within a single round trip; on subsequent connections between the same client and server, the client can often send application data immediately, that is, using a zero round-trip setup.

This document describes how TLS acts as a security component of QUIC.

2. Notational Conventions

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.

This document uses the terminology established in [QUIC-TRANSPORT].

For brevity, the acronym TLS is used to refer to TLS 1.3, though a newer version could be used; see Section 4.2.

2.1. TLS Overview

TLS provides two endpoints with a way to establish a means of communication over an untrusted medium (for example, the Internet). TLS enables authentication of peers and provides confidentiality and integrity protection for messages that endpoints exchange.

Internally, TLS is a layered protocol, with the structure shown in Figure 1.

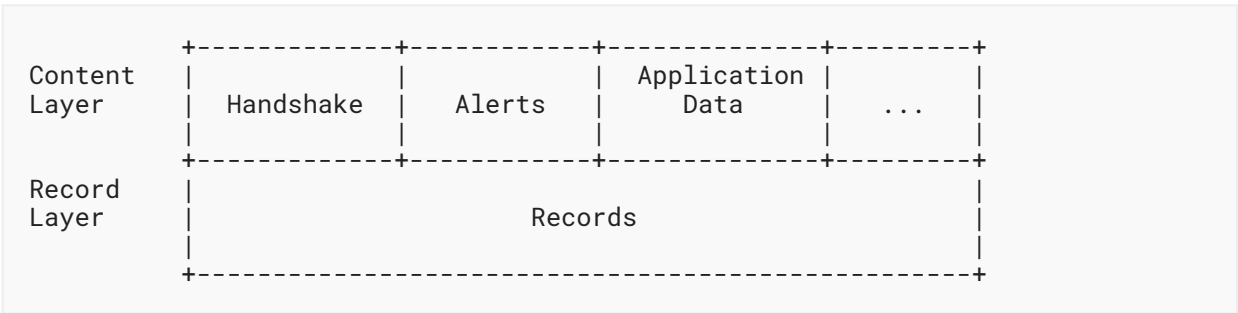

Figure 1: TLS Layers

Each content-layer message (e.g., handshake, alerts, and application data) is carried as a series of typed TLS records by the record layer. Records are individually cryptographically protected and then transmitted over a reliable transport (typically TCP), which provides sequencing and guaranteed delivery.

The TLS authenticated key exchange occurs between two endpoints: client and server. The client initiates the exchange and the server responds. If the key exchange completes successfully, both client and server will agree on a secret. TLS supports both pre-shared key (PSK) and Diffie-Hellman over either finite fields or elliptic curves ((EC)DHE) key exchanges. PSK is the basis for Early Data (0-RTT); the latter provides forward secrecy (FS) when the (EC)DHE keys are destroyed. The two modes can also be combined to provide forward secrecy while using the PSK for authentication.

After completing the TLS handshake, the client will have learned and authenticated an identity for the server, and the server is optionally able to learn and authenticate an identity for the client. TLS supports X.509 [RFC5280] certificate-based authentication for both server and client. When PSK key exchange is used (as in resumption), knowledge of the PSK serves to authenticate the peer.

The TLS key exchange is resistant to tampering by attackers, and it produces shared secrets that cannot be controlled by either participating peer.

TLS provides two basic handshake modes of interest to QUIC:

- A full 1-RTT handshake, in which the client is able to send application data after one round trip and the server immediately responds after receiving the first handshake message from the client.
- A 0-RTT handshake, in which the client uses information it has previously learned about the server to send application data immediately. This application data can be replayed by an attacker, so 0-RTT is not suitable for carrying instructions that might initiate any action that could cause unwanted effects if replayed.

A simplified TLS handshake with 0-RTT application data is shown in Figure 2.

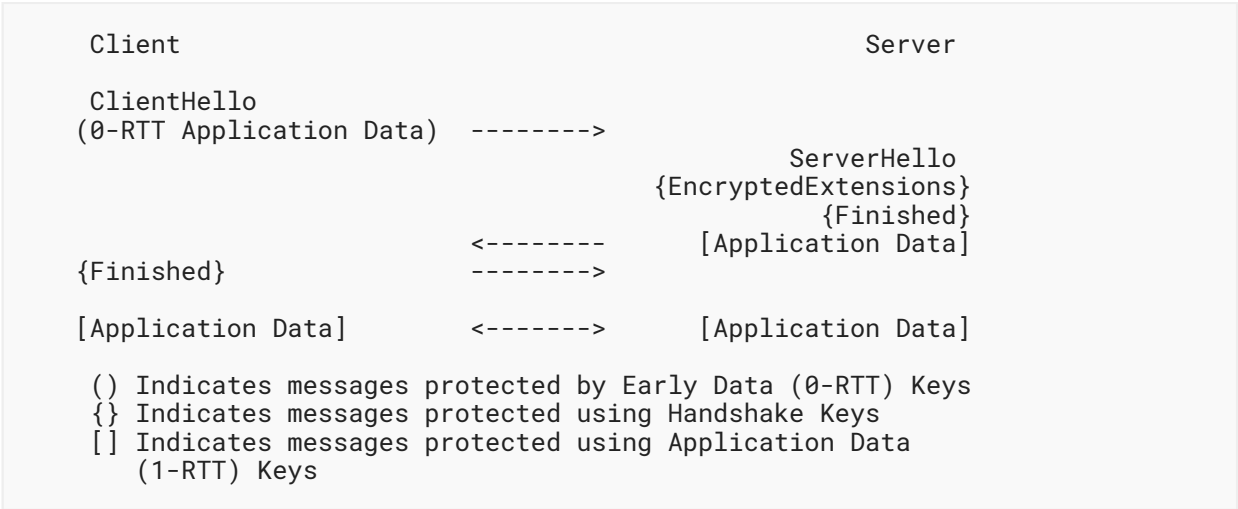

Figure 2: TLS Handshake with 0-RTT

Figure 2 omits the EndOfEarlyData message, which is not used in QUIC; see Section 8.3. Likewise, neither ChangeCipherSpec nor KeyUpdate messages are used by QUIC. ChangeCipherSpec is redundant in TLS 1.3; see Section 8.4. QUIC has its own key update mechanism; see Section 6.

Data is protected using a number of encryption levels:

- Initial keys
- Early data (0-RTT) keys
- Handshake keys

- Application data (1-RTT) keys

Application data can only appear in the early data and application data levels. Handshake and alert messages may appear in any level.

The 0-RTT handshake can be used if the client and server have previously communicated. In the 1-RTT handshake, the client is unable to send protected application data until it has received all of the handshake messages sent by the server.

3. Protocol Overview

QUIC [QUIC-TRANSPORT] assumes responsibility for the confidentiality and integrity protection of packets. For this it uses keys derived from a TLS handshake [TLS13], but instead of carrying TLS records over QUIC (as with TCP), TLS handshake and alert messages are carried directly over the QUIC transport, which takes over the responsibilities of the TLS record layer, as shown in Figure 3.

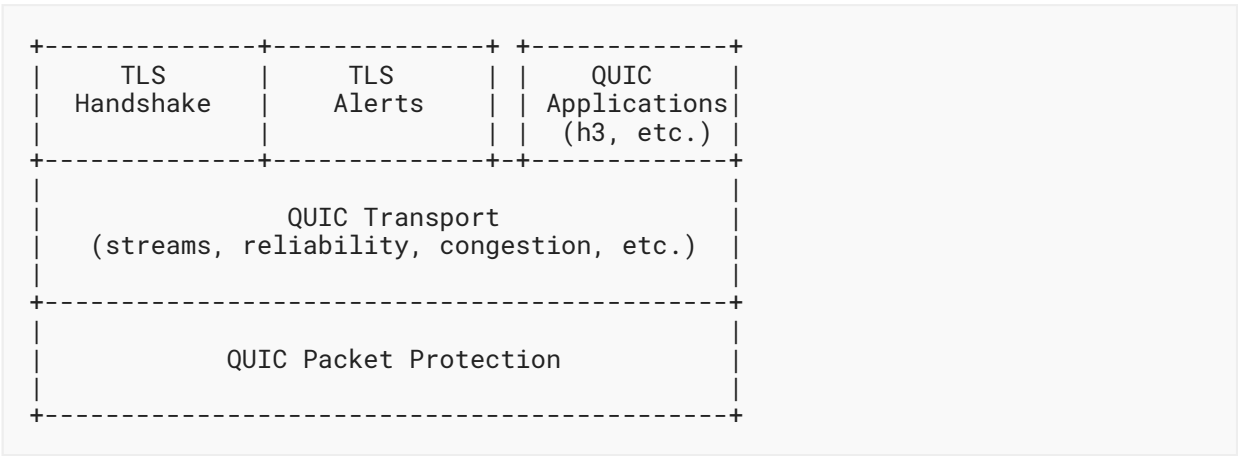

Figure 3: QUIC Layers

QUIC also relies on TLS for authentication and negotiation of parameters that are critical to security and performance.

Rather than a strict layering, these two protocols cooperate: QUIC uses the TLS handshake; TLS uses the reliability, ordered delivery, and record layer provided by QUIC.

At a high level, there are two main interactions between the TLS and QUIC components:

- The TLS component sends and receives messages via the QUIC component, with QUIC providing a reliable stream abstraction to TLS.
- The TLS component provides a series of updates to the QUIC component, including (a) new packet protection keys to install and (b) state changes such as handshake completion, the server certificate, etc.

Figure 4 shows these interactions in more detail, with the QUIC packet protection being called out specially.

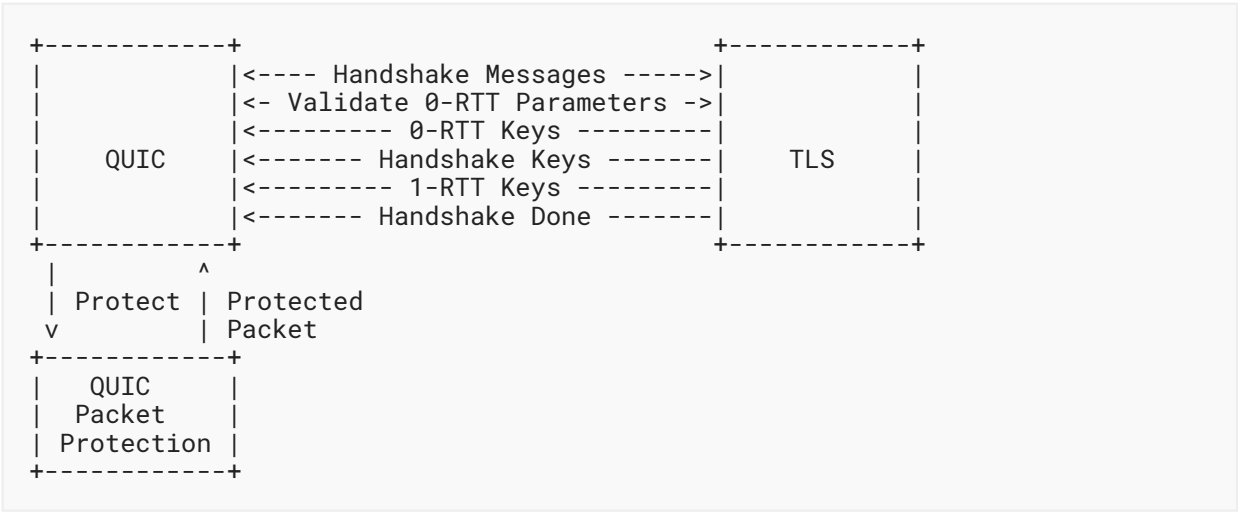

Figure 4: QUIC and TLS Interactions

Unlike TLS over TCP, QUIC applications that want to send data do not send it using TLS Application Data records. Rather, they send it as QUIC STREAM frames or other frame types, which are then carried in QUIC packets.

4. Carrying TLS Messages

QUIC carries TLS handshake data in CRYPTO frames, each of which consists of a contiguous block of handshake data identified by an offset and length. Those frames are packaged into QUIC packets and encrypted under the current encryption level. As with TLS over TCP, once TLS handshake data has been delivered to QUIC, it is QUIC's responsibility to deliver it reliably. Each chunk of data that is produced by TLS is associated with the set of keys that TLS is currently using. If QUIC needs to retransmit that data, it **MUST** use the same keys even if TLS has already updated to newer keys.

Each encryption level corresponds to a packet number space. The packet number space that is used determines the semantics of frames. Some frames are prohibited in different packet number spaces; see Section 12.5 of [QUIC-TRANSPORT].

Because packets could be reordered on the wire, QUIC uses the packet type to indicate which keys were used to protect a given packet, as shown in Table 1. When packets of different types need to be sent, endpoints **SHOULD** use coalesced packets to send them in the same UDP datagram.

Packet Type	Encryption Keys	PN Space
Initial	Initial secrets	Initial

Packet Type	Encryption Keys	PN Space
0-RTT Protected	0-RTT	Application data
Handshake	Handshake	Handshake
Retry	Retry	N/A
Version Negotiation	N/A	N/A
Short Header	1-RTT	Application data

Table 1: Encryption Keys by Packet Type

Section 17 of [QUIC-TRANSPORT] shows how packets at the various encryption levels fit into the handshake process.

4.1. Interface to TLS

As shown in Figure 4, the interface from QUIC to TLS consists of four primary functions:

- Sending and receiving handshake messages
- Processing stored transport and application state from a resumed session and determining if it is valid to generate or accept 0-RTT data
- Rekeying (both transmit and receive)
- Updating handshake state

Additional functions might be needed to configure TLS. In particular, QUIC and TLS need to agree on which is responsible for validation of peer credentials, such as certificate validation [RFC5280].

4.1.1. Handshake Complete

In this document, the TLS handshake is considered complete when the TLS stack has reported that the handshake is complete. This happens when the TLS stack has both sent a Finished message and verified the peer's Finished message. Verifying the peer's Finished message provides the endpoints with an assurance that previous handshake messages have not been modified. Note that the handshake does not complete at both endpoints simultaneously. Consequently, any requirement that is based on the completion of the handshake depends on the perspective of the endpoint in question.

4.1.2. Handshake Confirmed

In this document, the TLS handshake is considered confirmed at the server when the handshake completes. The server **MUST** send a HANDSHAKE_DONE frame as soon as the handshake is complete. At the client, the handshake is considered confirmed when a HANDSHAKE_DONE frame is received.

Additionally, a client **MAY** consider the handshake to be confirmed when it receives an acknowledgment for a 1-RTT packet. This can be implemented by recording the lowest packet number sent with 1-RTT keys and comparing it to the Largest Acknowledged field in any received 1-RTT ACK frame: once the latter is greater than or equal to the former, the handshake is confirmed.

4.1.3. Sending and Receiving Handshake Messages

In order to drive the handshake, TLS depends on being able to send and receive handshake messages. There are two basic functions on this interface: one where QUIC requests handshake messages and one where QUIC provides bytes that comprise handshake messages.

Before starting the handshake, QUIC provides TLS with the transport parameters (see Section 8.2) that it wishes to carry.

A QUIC client starts TLS by requesting TLS handshake bytes from TLS. The client acquires handshake bytes before sending its first packet. A QUIC server starts the process by providing TLS with the client's handshake bytes.

At any time, the TLS stack at an endpoint will have a current sending encryption level and a receiving encryption level. TLS encryption levels determine the QUIC packet type and keys that are used for protecting data.

Each encryption level is associated with a different sequence of bytes, which is reliably transmitted to the peer in CRYPTO frames. When TLS provides handshake bytes to be sent, they are appended to the handshake bytes for the current encryption level. The encryption level then determines the type of packet that the resulting CRYPTO frame is carried in; see Table 1.

Four encryption levels are used, producing keys for Initial, 0-RTT, Handshake, and 1-RTT packets. CRYPTO frames are carried in just three of these levels, omitting the 0-RTT level. These four levels correspond to three packet number spaces: Initial and Handshake encrypted packets use their own separate spaces; 0-RTT and 1-RTT packets use the application data packet number space.

QUIC takes the unprotected content of TLS handshake records as the content of CRYPTO frames. TLS record protection is not used by QUIC. QUIC assembles CRYPTO frames into QUIC packets, which are protected using QUIC packet protection.

QUIC CRYPTO frames only carry TLS handshake messages. TLS alerts are turned into QUIC CONNECTION_CLOSE error codes; see Section 4.8. TLS application data and other content types cannot be carried by QUIC at any encryption level; it is an error if they are received from the TLS stack.

When an endpoint receives a QUIC packet containing a CRYPTO frame from the network, it proceeds as follows:

- If the packet uses the current TLS receiving encryption level, sequence the data into the input flow as usual. As with STREAM frames, the offset is used to find the proper location in the data sequence. If the result of this process is that new data is available, then it is delivered to TLS in order.

- If the packet is from a previously installed encryption level, it **MUST NOT** contain data that extends past the end of previously received data in that flow. Implementations **MUST** treat any violations of this requirement as a connection error of type PROTOCOL_VIOLATION.
- If the packet is from a new encryption level, it is saved for later processing by TLS. Once TLS moves to receiving from this encryption level, saved data can be provided to TLS. When TLS provides keys for a higher encryption level, if there is data from a previous encryption level that TLS has not consumed, this **MUST** be treated as a connection error of type PROTOCOL_VIOLATION.

Each time that TLS is provided with new data, new handshake bytes are requested from TLS. TLS might not provide any bytes if the handshake messages it has received are incomplete or it has no data to send.

The content of CRYPTO frames might either be processed incrementally by TLS or buffered until complete messages or flights are available. TLS is responsible for buffering handshake bytes that have arrived in order. QUIC is responsible for buffering handshake bytes that arrive out of order or for encryption levels that are not yet ready. QUIC does not provide any means of flow control for CRYPTO frames; see Section 7.5 of [QUIC-TRANSPORT].

Once the TLS handshake is complete, this is indicated to QUIC along with any final handshake bytes that TLS needs to send. At this stage, the transport parameters that the peer advertised during the handshake are authenticated; see Section 8.2.

Once the handshake is complete, TLS becomes passive. TLS can still receive data from its peer and respond in kind, but it will not need to send more data unless specifically requested -- either by an application or QUIC. One reason to send data is that the server might wish to provide additional or updated session tickets to a client.

When the handshake is complete, QUIC only needs to provide TLS with any data that arrives in CRYPTO streams. In the same manner that is used during the handshake, new data is requested from TLS after providing received data.

4.1.4. Encryption Level Changes

As keys at a given encryption level become available to TLS, TLS indicates to QUIC that reading or writing keys at that encryption level are available.

The availability of new keys is always a result of providing inputs to TLS. TLS only provides new keys after being initialized (by a client) or when provided with new handshake data.

However, a TLS implementation could perform some of its processing asynchronously. In particular, the process of validating a certificate can take some time. While waiting for TLS processing to complete, an endpoint **SHOULD** buffer received packets if they might be processed using keys that are not yet available. These packets can be processed once keys are provided by TLS. An endpoint **SHOULD** continue to respond to packets that can be processed during this time.

After processing inputs, TLS might produce handshake bytes, keys for new encryption levels, or both.

TLS provides QUIC with three items as a new encryption level becomes available:

- A secret
- An Authenticated Encryption with Associated Data (AEAD) function
- A Key Derivation Function (KDF)

These values are based on the values that TLS negotiates and are used by QUIC to generate packet and header protection keys; see Section 5 and Section 5.4.

If 0-RTT is possible, it is ready after the client sends a TLS ClientHello message or the server receives that message. After providing a QUIC client with the first handshake bytes, the TLS stack might signal the change to 0-RTT keys. On the server, after receiving handshake bytes that contain a ClientHello message, a TLS server might signal that 0-RTT keys are available.

Although TLS only uses one encryption level at a time, QUIC may use more than one level. For instance, after sending its Finished message (using a CRYPTO frame at the Handshake encryption level) an endpoint can send STREAM data (in 1-RTT encryption). If the Finished message is lost, the endpoint uses the Handshake encryption level to retransmit the lost message. Reordering or loss of packets can mean that QUIC will need to handle packets at multiple encryption levels. During the handshake, this means potentially handling packets at higher and lower encryption levels than the current encryption level used by TLS.

In particular, server implementations need to be able to read packets at the Handshake encryption level at the same time as the 0-RTT encryption level. A client could interleave ACK frames that are protected with Handshake keys with 0-RTT data, and the server needs to process those acknowledgments in order to detect lost Handshake packets.

QUIC also needs access to keys that might not ordinarily be available to a TLS implementation. For instance, a client might need to acknowledge Handshake packets before it is ready to send CRYPTO frames at that encryption level. TLS therefore needs to provide keys to QUIC before it might produce them for its own use.

4.1.5. TLS Interface Summary

Figure 5 summarizes the exchange between QUIC and TLS for both client and server. Solid arrows indicate packets that carry handshake data; dashed arrows show where application data can be sent. Each arrow is tagged with the encryption level used for that transmission.

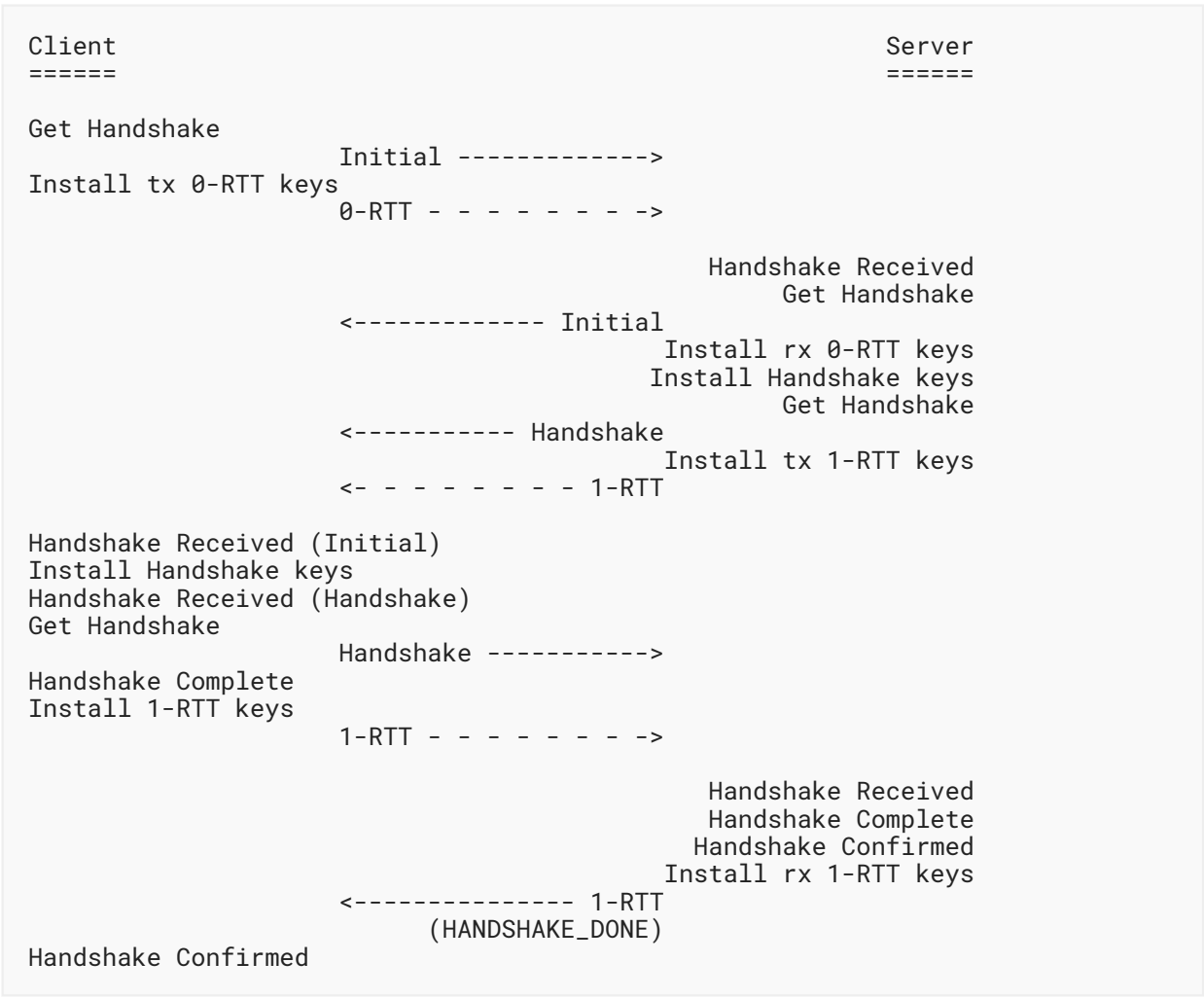

Figure 5: Interaction Summary between QUIC and TLS

Figure 5 shows the multiple packets that form a single "flight" of messages being processed individually, to show what incoming messages trigger different actions. This shows multiple "Get Handshake" invocations to retrieve handshake messages at different encryption levels. New handshake messages are requested after incoming packets have been processed.

Figure 5 shows one possible structure for a simple handshake exchange. The exact process varies based on the structure of endpoint implementations and the order in which packets arrive. Implementations could use a different number of operations or execute them in other orders.

4.2. TLS Version

This document describes how TLS 1.3 [TLS13] is used with QUIC.

In practice, the TLS handshake will negotiate a version of TLS to use. This could result in a version of TLS newer than 1.3 being negotiated if both endpoints support that version. This is acceptable provided that the features of TLS 1.3 that are used by QUIC are supported by the newer version.

Clients **MUST NOT** offer TLS versions older than 1.3. A badly configured TLS implementation could negotiate TLS 1.2 or another older version of TLS. An endpoint **MUST** terminate the connection if a version of TLS older than 1.3 is negotiated.

4.3. ClientHello Size

The first Initial packet from a client contains the start or all of its first cryptographic handshake message, which for TLS is the ClientHello. Servers might need to parse the entire ClientHello (e.g., to access extensions such as Server Name Identification (SNI) or Application-Layer Protocol Negotiation (ALPN)) in order to decide whether to accept the new incoming QUIC connection. If the ClientHello spans multiple Initial packets, such servers would need to buffer the first received fragments, which could consume excessive resources if the client's address has not yet been validated. To avoid this, servers **MAY** use the Retry feature (see Section 8.1 of [QUIC-TRANSPORT]) to only buffer partial ClientHello messages from clients with a validated address.

QUIC packet and framing add at least 36 bytes of overhead to the ClientHello message. That overhead increases if the client chooses a Source Connection ID field longer than zero bytes. Overheads also do not include the token or a Destination Connection ID longer than 8 bytes, both of which might be required if a server sends a Retry packet.

A typical TLS ClientHello can easily fit into a 1200-byte packet. However, in addition to the overheads added by QUIC, there are several variables that could cause this limit to be exceeded. Large session tickets, multiple or large key shares, and long lists of supported ciphers, signature algorithms, versions, QUIC transport parameters, and other negotiable parameters and extensions could cause this message to grow.

For servers, in addition to connection IDs and tokens, the size of TLS session tickets can have an effect on a client's ability to connect efficiently. Minimizing the size of these values increases the probability that clients can use them and still fit their entire ClientHello message in their first Initial packet.

The TLS implementation does not need to ensure that the ClientHello is large enough to meet QUIC's requirements for datagrams that carry Initial packets; see Section 14.1 of [QUIC-TRANSPORT]. QUIC implementations use PADDING frames or packet coalescing to ensure that datagrams are large enough.

4.4. Peer Authentication

The requirements for authentication depend on the application protocol that is in use. TLS provides server authentication and permits the server to request client authentication.

A client **MUST** authenticate the identity of the server. This typically involves verification that the identity of the server is included in a certificate and that the certificate is issued by a trusted entity (see for example [RFC2818]).

> Note: Where servers provide certificates for authentication, the size of the certificate chain can consume a large number of bytes. Controlling the size of certificate chains is critical to performance in QUIC as servers are limited to sending 3 bytes for every byte received prior to validating the client address; see Section 8.1 of [QUIC-TRANSPORT]. The size of a certificate chain can be managed by limiting the number of names or extensions; using keys with small public key representations, like ECDSA; or by using certificate compression [COMPRESS].

A server **MAY** request that the client authenticate during the handshake. A server **MAY** refuse a connection if the client is unable to authenticate when requested. The requirements for client authentication vary based on application protocol and deployment.

A server **MUST NOT** use post-handshake client authentication (as defined in Section 4.6.2 of [TLS13]) because the multiplexing offered by QUIC prevents clients from correlating the certificate request with the application-level event that triggered it (see [HTTP2-TLS13]). More specifically, servers **MUST NOT** send post-handshake TLS CertificateRequest messages, and clients **MUST** treat receipt of such messages as a connection error of type PROTOCOL_VIOLATION.

4.5. Session Resumption

QUIC can use the session resumption feature of TLS 1.3. It does this by carrying NewSessionTicket messages in CRYPTO frames after the handshake is complete. Session resumption can be used to provide 0-RTT and can also be used when 0-RTT is disabled.

Endpoints that use session resumption might need to remember some information about the current connection when creating a resumed connection. TLS requires that some information be retained; see Section 4.6.1 of [TLS13]. QUIC itself does not depend on any state being retained when resuming a connection unless 0-RTT is also used; see Section 7.4.1 of [QUIC-TRANSPORT] and Section 4.6.1. Application protocols could depend on state that is retained between resumed connections.

Clients can store any state required for resumption along with the session ticket. Servers can use the session ticket to help carry state.

Session resumption allows servers to link activity on the original connection with the resumed connection, which might be a privacy issue for clients. Clients can choose not to enable resumption to avoid creating this correlation. Clients **SHOULD NOT** reuse tickets as that allows entities other than the server to correlate connections; see Appendix C.4 of [TLS13].

4.6. 0-RTT

The 0-RTT feature in QUIC allows a client to send application data before the handshake is complete. This is made possible by reusing negotiated parameters from a previous connection. To enable this, 0-RTT depends on the client remembering critical parameters and providing the server with a TLS session ticket that allows the server to recover the same information.

This information includes parameters that determine TLS state, as governed by [TLS13], QUIC transport parameters, the chosen application protocol, and any information the application protocol might need; see Section 4.6.3. This information determines how 0-RTT packets and their contents are formed.

To ensure that the same information is available to both endpoints, all information used to establish 0-RTT comes from the same connection. Endpoints cannot selectively disregard information that might alter the sending or processing of 0-RTT.

[TLS13] sets a limit of seven days on the time between the original connection and any attempt to use 0-RTT. There are other constraints on 0-RTT usage, notably those caused by the potential exposure to replay attack; see Section 9.2.

4.6.1. Enabling 0-RTT

The TLS early_data extension in the NewSessionTicket message is defined to convey (in the max_early_data_size parameter) the amount of TLS 0-RTT data the server is willing to accept. QUIC does not use TLS early data. QUIC uses 0-RTT packets to carry early data. Accordingly, the max_early_data_size parameter is repurposed to hold a sentinel value 0xffffffff to indicate that the server is willing to accept QUIC 0-RTT data. To indicate that the server does not accept 0-RTT data, the early_data extension is omitted from the NewSessionTicket. The amount of data that the client can send in QUIC 0-RTT is controlled by the initial_max_data transport parameter supplied by the server.

Servers **MUST NOT** send the early_data extension with a max_early_data_size field set to any value other than 0xffffffff. A client **MUST** treat receipt of a NewSessionTicket that contains an early_data extension with any other value as a connection error of type PROTOCOL_VIOLATION.

A client that wishes to send 0-RTT packets uses the early_data extension in the ClientHello message of a subsequent handshake; see Section 4.2.10 of [TLS13]. It then sends application data in 0-RTT packets.

A client that attempts 0-RTT might also provide an address validation token if the server has sent a NEW_TOKEN frame; see Section 8.1 of [QUIC-TRANSPORT].

4.6.2. Accepting and Rejecting 0-RTT

A server accepts 0-RTT by sending an early_data extension in the EncryptedExtensions; see Section 4.2.10 of [TLS13]. The server then processes and acknowledges the 0-RTT packets that it receives.

A server rejects 0-RTT by sending the EncryptedExtensions without an early_data extension. A server will always reject 0-RTT if it sends a TLS HelloRetryRequest. When rejecting 0-RTT, a server **MUST NOT** process any 0-RTT packets, even if it could. When 0-RTT was rejected, a client **SHOULD** treat receipt of an acknowledgment for a 0-RTT packet as a connection error of type PROTOCOL_VIOLATION, if it is able to detect the condition.

When 0-RTT is rejected, all connection characteristics that the client assumed might be incorrect. This includes the choice of application protocol, transport parameters, and any application configuration. The client therefore **MUST** reset the state of all streams, including application state bound to those streams.

A client **MAY** reattempt 0-RTT if it receives a Retry or Version Negotiation packet. These packets do not signify rejection of 0-RTT.

4.6.3. Validating 0-RTT Configuration

When a server receives a ClientHello with the early_data extension, it has to decide whether to accept or reject 0-RTT data from the client. Some of this decision is made by the TLS stack (e.g., checking that the cipher suite being resumed was included in the ClientHello; see Section 4.2.10 of [TLS13]). Even when the TLS stack has no reason to reject 0-RTT data, the QUIC stack or the application protocol using QUIC might reject 0-RTT data because the configuration of the transport or application associated with the resumed session is not compatible with the server's current configuration.

QUIC requires additional transport state to be associated with a 0-RTT session ticket. One common way to implement this is using stateless session tickets and storing this state in the session ticket. Application protocols that use QUIC might have similar requirements regarding associating or storing state. This associated state is used for deciding whether 0-RTT data must be rejected. For example, HTTP/3 settings [QUIC-HTTP] determine how 0-RTT data from the client is interpreted. Other applications using QUIC could have different requirements for determining whether to accept or reject 0-RTT data.

4.7. HelloRetryRequest

The HelloRetryRequest message (see Section 4.1.4 of [TLS13]) can be used to request that a client provide new information, such as a key share, or to validate some characteristic of the client. From the perspective of QUIC, HelloRetryRequest is not differentiated from other cryptographic handshake messages that are carried in Initial packets. Although it is in principle possible to use this feature for address verification, QUIC implementations **SHOULD** instead use the Retry feature; see Section 8.1 of [QUIC-TRANSPORT].

4.8. TLS Errors

If TLS experiences an error, it generates an appropriate alert as defined in Section 6 of [TLS13].

A TLS alert is converted into a QUIC connection error. The AlertDescription value is added to 0x0100 to produce a QUIC error code from the range reserved for CRYPTO_ERROR; see Section 20.1 of [QUIC-TRANSPORT]. The resulting value is sent in a QUIC CONNECTION_CLOSE frame of type 0x1c.

QUIC is only able to convey an alert level of "fatal". In TLS 1.3, the only existing uses for the "warning" level are to signal connection close; see Section 6.1 of [TLS13]. As QUIC provides alternative mechanisms for connection termination and the TLS connection is only closed if an error is encountered, a QUIC endpoint **MUST** treat any alert from TLS as if it were at the "fatal" level.

QUIC permits the use of a generic code in place of a specific error code; see Section 11 of [QUIC-TRANSPORT]. For TLS alerts, this includes replacing any alert with a generic alert, such as handshake_failure (0x0128 in QUIC). Endpoints **MAY** use a generic error code to avoid possibly exposing confidential information.

4.9. Discarding Unused Keys

After QUIC has completed a move to a new encryption level, packet protection keys for previous encryption levels can be discarded. This occurs several times during the handshake, as well as when keys are updated; see Section 6.

Packet protection keys are not discarded immediately when new keys are available. If packets from a lower encryption level contain CRYPTO frames, frames that retransmit that data **MUST** be sent at the same encryption level. Similarly, an endpoint generates acknowledgments for packets at the same encryption level as the packet being acknowledged. Thus, it is possible that keys for a lower encryption level are needed for a short time after keys for a newer encryption level are available.

An endpoint cannot discard keys for a given encryption level unless it has received all the cryptographic handshake messages from its peer at that encryption level and its peer has done the same. Different methods for determining this are provided for Initial keys (Section 4.9.1) and Handshake keys (Section 4.9.2). These methods do not prevent packets from being received or sent at that encryption level because a peer might not have received all the acknowledgments necessary.

Though an endpoint might retain older keys, new data **MUST** be sent at the highest currently available encryption level. Only ACK frames and retransmissions of data in CRYPTO frames are sent at a previous encryption level. These packets **MAY** also include PADDING frames.

4.9.1. Discarding Initial Keys

Packets protected with Initial secrets (Section 5.2) are not authenticated, meaning that an attacker could spoof packets with the intent to disrupt a connection. To limit these attacks, Initial packet protection keys are discarded more aggressively than other keys.

The successful use of Handshake packets indicates that no more Initial packets need to be exchanged, as these keys can only be produced after receiving all CRYPTO frames from Initial packets. Thus, a client **MUST** discard Initial keys when it first sends a Handshake packet and a server **MUST** discard Initial keys when it first successfully processes a Handshake packet. Endpoints **MUST NOT** send Initial packets after this point.

This results in abandoning loss recovery state for the Initial encryption level and ignoring any outstanding Initial packets.

4.9.2. Discarding Handshake Keys

An endpoint **MUST** discard its Handshake keys when the TLS handshake is confirmed (Section 4.1.2).

4.9.3. Discarding 0-RTT Keys

0-RTT and 1-RTT packets share the same packet number space, and clients do not send 0-RTT packets after sending a 1-RTT packet (Section 5.6).

Therefore, a client **SHOULD** discard 0-RTT keys as soon as it installs 1-RTT keys as they have no use after that moment.

Additionally, a server **MAY** discard 0-RTT keys as soon as it receives a 1-RTT packet. However, due to packet reordering, a 0-RTT packet could arrive after a 1-RTT packet. Servers **MAY** temporarily retain 0-RTT keys to allow decrypting reordered packets without requiring their contents to be retransmitted with 1-RTT keys. After receiving a 1-RTT packet, servers **MUST** discard 0-RTT keys within a short time; the **RECOMMENDED** time period is three times the Probe Timeout (PTO, see [QUIC-RECOVERY]). A server **MAY** discard 0-RTT keys earlier if it determines that it has received all 0-RTT packets, which can be done by keeping track of missing packet numbers.

5. Packet Protection

As with TLS over TCP, QUIC protects packets with keys derived from the TLS handshake, using the AEAD algorithm [AEAD] negotiated by TLS.

QUIC packets have varying protections depending on their type:

- Version Negotiation packets have no cryptographic protection.
- Retry packets use AEAD_AES_128_GCM to provide protection against accidental modification and to limit the entities that can produce a valid Retry; see Section 5.8.
- Initial packets use AEAD_AES_128_GCM with keys derived from the Destination Connection ID field of the first Initial packet sent by the client; see Section 5.2.
- All other packets have strong cryptographic protections for confidentiality and integrity, using keys and algorithms negotiated by TLS.

This section describes how packet protection is applied to Handshake packets, 0-RTT packets, and 1-RTT packets. The same packet protection process is applied to Initial packets. However, as it is trivial to determine the keys used for Initial packets, these packets are not considered to have confidentiality or integrity protection. Retry packets use a fixed key and so similarly lack confidentiality and integrity protection.

5.1. Packet Protection Keys

QUIC derives packet protection keys in the same way that TLS derives record protection keys.

Each encryption level has separate secret values for protection of packets sent in each direction. These traffic secrets are derived by TLS (see Section 7.1 of [TLS13]) and are used by QUIC for all encryption levels except the Initial encryption level. The secrets for the Initial encryption level are computed based on the client's initial Destination Connection ID, as described in Section 5.2.

The keys used for packet protection are computed from the TLS secrets using the KDF provided by TLS. In TLS 1.3, the HKDF-Expand-Label function described in Section 7.1 of [TLS13] is used with the hash function from the negotiated cipher suite. All uses of HKDF-Expand-Label in QUIC use a zero-length Context.

Note that labels, which are described using strings, are encoded as bytes using ASCII [ASCII] without quotes or any trailing NUL byte.

Other versions of TLS **MUST** provide a similar function in order to be used with QUIC.

The current encryption level secret and the label "quic key" are input to the KDF to produce the AEAD key; the label "quic iv" is used to derive the Initialization Vector (IV); see Section 5.3. The header protection key uses the "quic hp" label; see Section 5.4. Using these labels provides key separation between QUIC and TLS; see Section 9.6.

Both "quic key" and "quic hp" are used to produce keys, so the Length provided to HKDF-Expand-Label along with these labels is determined by the size of keys in the AEAD or header protection algorithm. The Length provided with "quic iv" is the minimum length of the AEAD nonce or 8 bytes if that is larger; see [AEAD].

The KDF used for initial secrets is always the HKDF-Expand-Label function from TLS 1.3; see Section 5.2.

5.2. Initial Secrets

Initial packets apply the packet protection process, but use a secret derived from the Destination Connection ID field from the client's first Initial packet.

This secret is determined by using HKDF-Extract (see Section 2.2 of [HKDF]) with a salt of 0x38762cf7f55934b34d179ae6a4c80cadccbb7f0a and the input keying material (IKM) of the Destination Connection ID field. This produces an intermediate pseudorandom key (PRK) that is used to derive two separate secrets for sending and receiving.

The secret used by clients to construct Initial packets uses the PRK and the label "client in" as input to the HKDF-Expand-Label function from TLS [TLS13] to produce a 32-byte secret. Packets constructed by the server use the same process with the label "server in". The hash function for HKDF when deriving initial secrets and keys is SHA-256 [SHA].

This process in pseudocode is:

```
initial_salt = 0x38762cf7f55934b34d179ae6a4c80cadccbb7f0a
initial_secret = HKDF-Extract(initial_salt,
                              client_dst_connection_id)

client_initial_secret = HKDF-Expand-Label(initial_secret,
                                          "client in", "",
                                          Hash.length)
server_initial_secret = HKDF-Expand-Label(initial_secret,
                                          "server in", "",
                                          Hash.length)
```

The connection ID used with HKDF-Expand-Label is the Destination Connection ID in the Initial packet sent by the client. This will be a randomly selected value unless the client creates the Initial packet after receiving a Retry packet, where the Destination Connection ID is selected by the server.

Future versions of QUIC **SHOULD** generate a new salt value, thus ensuring that the keys are different for each version of QUIC. This prevents a middlebox that recognizes only one version of QUIC from seeing or modifying the contents of packets from future versions.

The HKDF-Expand-Label function defined in TLS 1.3 **MUST** be used for Initial packets even where the TLS versions offered do not include TLS 1.3.

The secrets used for constructing subsequent Initial packets change when a server sends a Retry packet to use the connection ID value selected by the server. The secrets do not change when a client changes the Destination Connection ID it uses in response to an Initial packet from the server.

> Note: The Destination Connection ID field could be any length up to 20 bytes, including zero length if the server sends a Retry packet with a zero-length Source Connection ID field. After a Retry, the Initial keys provide the client no assurance that the server received its packet, so the client has to rely on the exchange that included the Retry packet to validate the server address; see Section 8.1 of [QUIC-TRANSPORT].

Appendix A contains sample Initial packets.

5.3. AEAD Usage

The Authenticated Encryption with Associated Data (AEAD) function (see [AEAD]) used for QUIC packet protection is the AEAD that is negotiated for use with the TLS connection. For example, if TLS is using the TLS_AES_128_GCM_SHA256 cipher suite, the AEAD_AES_128_GCM function is used.

QUIC can use any of the cipher suites defined in [TLS13] with the exception of TLS_AES_128_CCM_8_SHA256. A cipher suite **MUST NOT** be negotiated unless a header protection scheme is defined for the cipher suite. This document defines a header protection scheme for all cipher suites defined in [TLS13] aside from TLS_AES_128_CCM_8_SHA256. These cipher suites have a 16-byte authentication tag and produce an output 16 bytes larger than their input.

An endpoint **MUST NOT** reject a ClientHello that offers a cipher suite that it does not support, or it would be impossible to deploy a new cipher suite. This also applies to TLS_AES_128_CCM_8_SHA256.

When constructing packets, the AEAD function is applied prior to applying header protection; see Section 5.4. The unprotected packet header is part of the associated data (A). When processing packets, an endpoint first removes the header protection.

The key and IV for the packet are computed as described in Section 5.1. The nonce, N, is formed by combining the packet protection IV with the packet number. The 62 bits of the reconstructed QUIC packet number in network byte order are left-padded with zeros to the size of the IV. The exclusive OR of the padded packet number and the IV forms the AEAD nonce.

The associated data, A, for the AEAD is the contents of the QUIC header, starting from the first byte of either the short or long header, up to and including the unprotected packet number.

The input plaintext, P, for the AEAD is the payload of the QUIC packet, as described in [QUIC-TRANSPORT].

The output ciphertext, C, of the AEAD is transmitted in place of P.

Some AEAD functions have limits for how many packets can be encrypted under the same key and IV; see Section 6.6. This might be lower than the packet number limit. An endpoint **MUST** initiate a key update (Section 6) prior to exceeding any limit set for the AEAD that is in use.

5.4. Header Protection

Parts of QUIC packet headers, in particular the Packet Number field, are protected using a key that is derived separately from the packet protection key and IV. The key derived using the "quic hp" label is used to provide confidentiality protection for those fields that are not exposed to on-path elements.

This protection applies to the least significant bits of the first byte, plus the Packet Number field. The four least significant bits of the first byte are protected for packets with long headers; the five least significant bits of the first byte are protected for packets with short headers. For both header forms, this covers the reserved bits and the Packet Number Length field; the Key Phase bit is also protected for packets with a short header.

The same header protection key is used for the duration of the connection, with the value not changing after a key update (see Section 6). This allows header protection to be used to protect the key phase.

This process does not apply to Retry or Version Negotiation packets, which do not contain a protected payload or any of the fields that are protected by this process.

5.4.1. Header Protection Application

Header protection is applied after packet protection is applied (see Section 5.3). The ciphertext of the packet is sampled and used as input to an encryption algorithm. The algorithm used depends on the negotiated AEAD.

The output of this algorithm is a 5-byte mask that is applied to the protected header fields using exclusive OR. The least significant bits of the first byte of the packet are masked by the least significant bits of the first mask byte, and the packet number is masked with the remaining bytes. Any unused bytes of mask that might result from a shorter packet number encoding are unused.

Figure 6 shows a sample algorithm for applying header protection. Removing header protection only differs in the order in which the packet number length (pn_length) is determined (here "^" is used to represent exclusive OR).

```
mask = header_protection(hp_key, sample)

pn_length = (packet[0] & 0x03) + 1
if (packet[0] & 0x80) == 0x80:
   # Long header: 4 bits masked
   packet[0] ^= mask[0] & 0x0f
else:
   # Short header: 5 bits masked
   packet[0] ^= mask[0] & 0x1f

# pn_offset is the start of the Packet Number field.
packet[pn_offset:pn_offset+pn_length] ^= mask[1:1+pn_length]
```

Figure 6: Header Protection Pseudocode

Specific header protection functions are defined based on the selected cipher suite; see Section 5.4.3 and Section 5.4.4.

Figure 7 shows an example long header packet (Initial) and a short header packet (1-RTT). Figure 7 shows the fields in each header that are covered by header protection and the portion of the protected packet payload that is sampled.

```
Initial Packet {
  Header Form (1) = 1,
  Fixed Bit (1) = 1,
  Long Packet Type (2) = 0,
  Reserved Bits (2),         # Protected
  Packet Number Length (2),  # Protected
  Version (32),
  DCID Len (8),
  Destination Connection ID (0..160),
  SCID Len (8),
  Source Connection ID (0..160),
  Token Length (i),
  Token (..),
  Length (i),
  Packet Number (8..32),     # Protected
  Protected Payload (0..24), # Skipped Part
  Protected Payload (128),   # Sampled Part
  Protected Payload (..)     # Remainder
}

1-RTT Packet {
  Header Form (1) = 0,
  Fixed Bit (1) = 1,
  Spin Bit (1),
  Reserved Bits (2),         # Protected
  Key Phase (1),             # Protected
  Packet Number Length (2),  # Protected
  Destination Connection ID (0..160),
  Packet Number (8..32),     # Protected
  Protected Payload (0..24), # Skipped Part
  Protected Payload (128),   # Sampled Part
  Protected Payload (..),    # Remainder
}
```

Figure 7: Header Protection and Ciphertext Sample

Before a TLS cipher suite can be used with QUIC, a header protection algorithm **MUST** be specified for the AEAD used with that cipher suite. This document defines algorithms for AEAD_AES_128_GCM, AEAD_AES_128_CCM, AEAD_AES_256_GCM (all these AES AEADs are defined in [AEAD]), and AEAD_CHACHA20_POLY1305 (defined in [CHACHA]). Prior to TLS selecting a cipher suite, AES header protection is used (Section 5.4.3), matching the AEAD_AES_128_GCM packet protection.

5.4.2. Header Protection Sample

The header protection algorithm uses both the header protection key and a sample of the ciphertext from the packet Payload field.

The same number of bytes are always sampled, but an allowance needs to be made for the removal of protection by a receiving endpoint, which will not know the length of the Packet Number field. The sample of ciphertext is taken starting from an offset of 4 bytes after the start of the Packet Number field. That is, in sampling packet ciphertext for header protection, the Packet Number field is assumed to be 4 bytes long (its maximum possible encoded length).

An endpoint **MUST** discard packets that are not long enough to contain a complete sample.

To ensure that sufficient data is available for sampling, packets are padded so that the combined lengths of the encoded packet number and protected payload is at least 4 bytes longer than the sample required for header protection. The cipher suites defined in [TLS13] -- other than TLS_AES_128_CCM_8_SHA256, for which a header protection scheme is not defined in this document -- have 16-byte expansions and 16-byte header protection samples. This results in needing at least 3 bytes of frames in the unprotected payload if the packet number is encoded on a single byte, or 2 bytes of frames for a 2-byte packet number encoding.

The sampled ciphertext can be determined by the following pseudocode:

```
# pn_offset is the start of the Packet Number field.
sample_offset = pn_offset + 4

sample = packet[sample_offset..sample_offset+sample_length]
```

Where the packet number offset of a short header packet can be calculated as:

```
pn_offset = 1 + len(connection_id)
```

And the packet number offset of a long header packet can be calculated as:

```
pn_offset = 7 + len(destination_connection_id) +
                len(source_connection_id) +
                len(payload_length)
if packet_type == Initial:
    pn_offset += len(token_length) +
                 len(token)
```

For example, for a packet with a short header, an 8-byte connection ID, and protected with AEAD_AES_128_GCM, the sample takes bytes 13 to 28 inclusive (using zero-based indexing).

Multiple QUIC packets might be included in the same UDP datagram. Each packet is handled separately.

5.4.3. AES-Based Header Protection

This section defines the packet protection algorithm for AEAD_AES_128_GCM, AEAD_AES_128_CCM, and AEAD_AES_256_GCM. AEAD_AES_128_GCM and AEAD_AES_128_CCM use 128-bit AES in Electronic Codebook (ECB) mode. AEAD_AES_256_GCM uses 256-bit AES in ECB mode. AES is defined in [AES].

This algorithm samples 16 bytes from the packet ciphertext. This value is used as the input to AES-ECB. In pseudocode, the header protection function is defined as:

```
header_protection(hp_key, sample):
  mask = AES-ECB(hp_key, sample)
```

5.4.4. ChaCha20-Based Header Protection

When AEAD_CHACHA20_POLY1305 is in use, header protection uses the raw ChaCha20 function as defined in Section 2.4 of [CHACHA]. This uses a 256-bit key and 16 bytes sampled from the packet protection output.

The first 4 bytes of the sampled ciphertext are the block counter. A ChaCha20 implementation could take a 32-bit integer in place of a byte sequence, in which case, the byte sequence is interpreted as a little-endian value.

The remaining 12 bytes are used as the nonce. A ChaCha20 implementation might take an array of three 32-bit integers in place of a byte sequence, in which case, the nonce bytes are interpreted as a sequence of 32-bit little-endian integers.

The encryption mask is produced by invoking ChaCha20 to protect 5 zero bytes. In pseudocode, the header protection function is defined as:

```
header_protection(hp_key, sample):
  counter = sample[0..3]
  nonce = sample[4..15]
  mask = ChaCha20(hp_key, counter, nonce, {0,0,0,0,0})
```

5.5. Receiving Protected Packets

Once an endpoint successfully receives a packet with a given packet number, it **MUST** discard all packets in the same packet number space with higher packet numbers if they cannot be successfully unprotected with either the same key, or -- if there is a key update -- a subsequent packet protection key; see Section 6. Similarly, a packet that appears to trigger a key update but cannot be unprotected successfully **MUST** be discarded.

Failure to unprotect a packet does not necessarily indicate the existence of a protocol error in a peer or an attack. The truncated packet number encoding used in QUIC can cause packet numbers to be decoded incorrectly if they are delayed significantly.

5.6. Use of 0-RTT Keys

If 0-RTT keys are available (see Section 4.6.1), the lack of replay protection means that restrictions on their use are necessary to avoid replay attacks on the protocol.

Of the frames defined in [QUIC-TRANSPORT], the STREAM, RESET_STREAM, STOP_SENDING, and CONNECTION_CLOSE frames are potentially unsafe for use with 0-RTT as they carry application data. Application data that is received in 0-RTT could cause an application at the server to process the data multiple times rather than just once. Additional actions taken by a server as a result of processing replayed application data could have unwanted consequences. A client therefore **MUST NOT** use 0-RTT for application data unless specifically requested by the application that is in use.

An application protocol that uses QUIC **MUST** include a profile that defines acceptable use of 0-RTT; otherwise, 0-RTT can only be used to carry QUIC frames that do not carry application data. For example, a profile for HTTP is described in [HTTP-REPLAY] and used for HTTP/3; see Section 10.9 of [QUIC-HTTP].

Though replaying packets might result in additional connection attempts, the effect of processing replayed frames that do not carry application data is limited to changing the state of the affected connection. A TLS handshake cannot be successfully completed using replayed packets.

A client **MAY** wish to apply additional restrictions on what data it sends prior to the completion of the TLS handshake.

A client otherwise treats 0-RTT keys as equivalent to 1-RTT keys, except that it cannot send certain frames with 0-RTT keys; see Section 12.5 of [QUIC-TRANSPORT].

A client that receives an indication that its 0-RTT data has been accepted by a server can send 0-RTT data until it receives all of the server's handshake messages. A client **SHOULD** stop sending 0-RTT data if it receives an indication that 0-RTT data has been rejected.

A server **MUST NOT** use 0-RTT keys to protect packets; it uses 1-RTT keys to protect acknowledgments of 0-RTT packets. A client **MUST NOT** attempt to decrypt 0-RTT packets it receives and instead **MUST** discard them.

Once a client has installed 1-RTT keys, it **MUST NOT** send any more 0-RTT packets.

> Note: 0-RTT data can be acknowledged by the server as it receives it, but any packets containing acknowledgments of 0-RTT data cannot have packet protection removed by the client until the TLS handshake is complete. The 1-RTT keys necessary to remove packet protection cannot be derived until the client receives all server handshake messages.

5.7. Receiving Out-of-Order Protected Packets

Due to reordering and loss, protected packets might be received by an endpoint before the final TLS handshake messages are received. A client will be unable to decrypt 1-RTT packets from the server, whereas a server will be able to decrypt 1-RTT packets from the client. Endpoints in either role **MUST NOT** decrypt 1-RTT packets from their peer prior to completing the handshake.

Even though 1-RTT keys are available to a server after receiving the first handshake messages from a client, it is missing assurances on the client state:

- The client is not authenticated, unless the server has chosen to use a pre-shared key and validated the client's pre-shared key binder; see Section 4.2.11 of [TLS13].
- The client has not demonstrated liveness, unless the server has validated the client's address with a Retry packet or other means; see Section 8.1 of [QUIC-TRANSPORT].
- Any received 0-RTT data that the server responds to might be due to a replay attack.

Therefore, the server's use of 1-RTT keys before the handshake is complete is limited to sending data. A server **MUST NOT** process incoming 1-RTT protected packets before the TLS handshake is complete. Because sending acknowledgments indicates that all frames in a packet have been processed, a server cannot send acknowledgments for 1-RTT packets until the TLS handshake is complete. Received packets protected with 1-RTT keys **MAY** be stored and later decrypted and used once the handshake is complete.

> Note: TLS implementations might provide all 1-RTT secrets prior to handshake completion. Even where QUIC implementations have 1-RTT read keys, those keys are not to be used prior to completing the handshake.

The requirement for the server to wait for the client Finished message creates a dependency on that message being delivered. A client can avoid the potential for head-of-line blocking that this implies by sending its 1-RTT packets coalesced with a Handshake packet containing a copy of the CRYPTO frame that carries the Finished message, until one of the Handshake packets is acknowledged. This enables immediate server processing for those packets.

A server could receive packets protected with 0-RTT keys prior to receiving a TLS ClientHello. The server **MAY** retain these packets for later decryption in anticipation of receiving a ClientHello.

A client generally receives 1-RTT keys at the same time as the handshake completes. Even if it has 1-RTT secrets, a client **MUST NOT** process incoming 1-RTT protected packets before the TLS handshake is complete.

5.8. Retry Packet Integrity

Retry packets (see Section 17.2.5 of [QUIC-TRANSPORT]) carry a Retry Integrity Tag that provides two properties: it allows the discarding of packets that have accidentally been corrupted by the network, and only an entity that observes an Initial packet can send a valid Retry packet.

The Retry Integrity Tag is a 128-bit field that is computed as the output of AEAD_AES_128_GCM [AEAD] used with the following inputs:

- The secret key, K, is 128 bits equal to 0xbe0c690b9f66575a1d766b54e368c84e.
- The nonce, N, is 96 bits equal to 0x461599d35d632bf2239825bb.
- The plaintext, P, is empty.

- The associated data, A, is the contents of the Retry Pseudo-Packet, as illustrated in Figure 8:

The secret key and the nonce are values derived by calling HKDF-Expand-Label using 0xd9c9943e6101fd200021506bcc02814c73030f25c79d71ce876eca876e6fca8e as the secret, with labels being "quic key" and "quic iv" (Section 5.1).

```
Retry Pseudo-Packet {
  ODCID Length (8),
  Original Destination Connection ID (0..160),
  Header Form (1) = 1,
  Fixed Bit (1) = 1,
  Long Packet Type (2) = 3,
  Unused (4),
  Version (32),
  DCID Len (8),
  Destination Connection ID (0..160),
  SCID Len (8),
  Source Connection ID (0..160),
  Retry Token (..),
}
```

Figure 8: Retry Pseudo-Packet

The Retry Pseudo-Packet is not sent over the wire. It is computed by taking the transmitted Retry packet, removing the Retry Integrity Tag, and prepending the two following fields:

ODCID Length: The ODCID Length field contains the length in bytes of the Original Destination Connection ID field that follows it, encoded as an 8-bit unsigned integer.

Original Destination Connection ID: The Original Destination Connection ID contains the value of the Destination Connection ID from the Initial packet that this Retry is in response to. The length of this field is given in ODCID Length. The presence of this field ensures that a valid Retry packet can only be sent by an entity that observes the Initial packet.

6. Key Update

Once the handshake is confirmed (see Section 4.1.2), an endpoint **MAY** initiate a key update.

The Key Phase bit indicates which packet protection keys are used to protect the packet. The Key Phase bit is initially set to 0 for the first set of 1-RTT packets and toggled to signal each subsequent key update.

The Key Phase bit allows a recipient to detect a change in keying material without needing to receive the first packet that triggered the change. An endpoint that notices a changed Key Phase bit updates keys and decrypts the packet that contains the changed value.

Initiating a key update results in both endpoints updating keys. This differs from TLS where endpoints can update keys independently.

This mechanism replaces the key update mechanism of TLS, which relies on KeyUpdate messages sent using 1-RTT encryption keys. Endpoints **MUST NOT** send a TLS KeyUpdate message. Endpoints **MUST** treat the receipt of a TLS KeyUpdate message as a connection error of type 0x010a, equivalent to a fatal TLS alert of unexpected_message; see Section 4.8.

Figure 9 shows a key update process, where the initial set of keys used (identified with @M) are replaced by updated keys (identified with @N). The value of the Key Phase bit is indicated in brackets [].

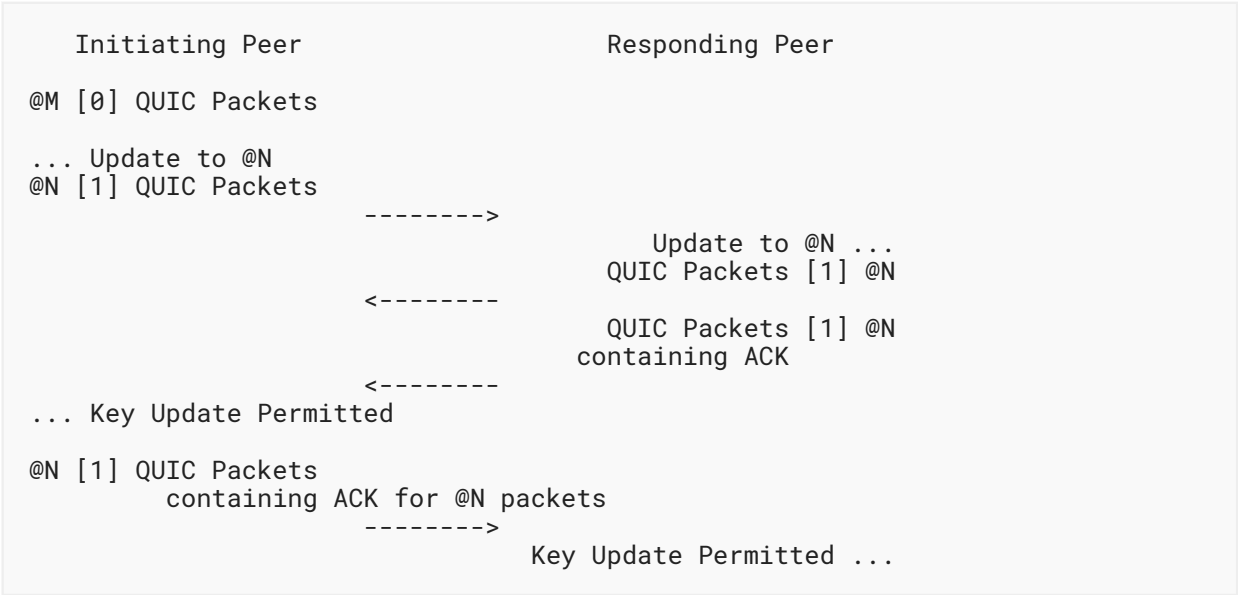

Figure 9: Key Update

6.1. Initiating a Key Update

Endpoints maintain separate read and write secrets for packet protection. An endpoint initiates a key update by updating its packet protection write secret and using that to protect new packets. The endpoint creates a new write secret from the existing write secret as performed in Section 7.2 of [TLS13]. This uses the KDF function provided by TLS with a label of "quic ku". The corresponding key and IV are created from that secret as defined in Section 5.1. The header protection key is not updated.

For example, to update write keys with TLS 1.3, HKDF-Expand-Label is used as:

```
secret_<n+1> = HKDF-Expand-Label(secret_<n>, "quic ku",
                                 "", Hash.length)
```

The endpoint toggles the value of the Key Phase bit and uses the updated key and IV to protect all subsequent packets.

An endpoint **MUST NOT** initiate a key update prior to having confirmed the handshake (Section 4.1.2). An endpoint **MUST NOT** initiate a subsequent key update unless it has received an acknowledgment for a packet that was sent protected with keys from the current key phase. This ensures that keys are available to both peers before another key update can be initiated. This can be implemented by tracking the lowest packet number sent with each key phase and the highest acknowledged packet number in the 1-RTT space: once the latter is higher than or equal to the former, another key update can be initiated.

> Note: Keys of packets other than the 1-RTT packets are never updated; their keys are derived solely from the TLS handshake state.

The endpoint that initiates a key update also updates the keys that it uses for receiving packets. These keys will be needed to process packets the peer sends after updating.

An endpoint **MUST** retain old keys until it has successfully unprotected a packet sent using the new keys. An endpoint **SHOULD** retain old keys for some time after unprotecting a packet sent using the new keys. Discarding old keys too early can cause delayed packets to be discarded. Discarding packets will be interpreted as packet loss by the peer and could adversely affect performance.

6.2. Responding to a Key Update

A peer is permitted to initiate a key update after receiving an acknowledgment of a packet in the current key phase. An endpoint detects a key update when processing a packet with a key phase that differs from the value used to protect the last packet it sent. To process this packet, the endpoint uses the next packet protection key and IV. See Section 6.3 for considerations about generating these keys.

If a packet is successfully processed using the next key and IV, then the peer has initiated a key update. The endpoint **MUST** update its send keys to the corresponding key phase in response, as described in Section 6.1. Sending keys **MUST** be updated before sending an acknowledgment for the packet that was received with updated keys. By acknowledging the packet that triggered the key update in a packet protected with the updated keys, the endpoint signals that the key update is complete.

An endpoint can defer sending the packet or acknowledgment according to its normal packet sending behavior; it is not necessary to immediately generate a packet in response to a key update. The next packet sent by the endpoint will use the updated keys. The next packet that contains an acknowledgment will cause the key update to be completed. If an endpoint detects a second update before it has sent any packets with updated keys containing an acknowledgment for the packet that initiated the key update, it indicates that its peer has updated keys twice without awaiting confirmation. An endpoint **MAY** treat such consecutive key updates as a connection error of type KEY_UPDATE_ERROR.

An endpoint that receives an acknowledgment that is carried in a packet protected with old keys where any acknowledged packet was protected with newer keys **MAY** treat that as a connection error of type KEY_UPDATE_ERROR. This indicates that a peer has received and acknowledged a packet that initiates a key update, but has not updated keys in response.

6.3. Timing of Receive Key Generation

Endpoints responding to an apparent key update **MUST NOT** generate a timing side-channel signal that might indicate that the Key Phase bit was invalid (see Section 9.5). Endpoints can use randomized packet protection keys in place of discarded keys when key updates are not yet permitted. Using randomized keys ensures that attempting to remove packet protection does not result in timing variations, and results in packets with an invalid Key Phase bit being rejected.

The process of creating new packet protection keys for receiving packets could reveal that a key update has occurred. An endpoint **MAY** generate new keys as part of packet processing, but this creates a timing signal that could be used by an attacker to learn when key updates happen and thus leak the value of the Key Phase bit.

Endpoints are generally expected to have current and next receive packet protection keys available. For a short period after a key update completes, up to the PTO, endpoints **MAY** defer generation of the next set of receive packet protection keys. This allows endpoints to retain only two sets of receive keys; see Section 6.5.

Once generated, the next set of packet protection keys **SHOULD** be retained, even if the packet that was received was subsequently discarded. Packets containing apparent key updates are easy to forge, and while the process of key update does not require significant effort, triggering this process could be used by an attacker for DoS.

For this reason, endpoints **MUST** be able to retain two sets of packet protection keys for receiving packets: the current and the next. Retaining the previous keys in addition to these might improve performance, but this is not essential.

6.4. Sending with Updated Keys

An endpoint never sends packets that are protected with old keys. Only the current keys are used. Keys used for protecting packets can be discarded immediately after switching to newer keys.

Packets with higher packet numbers **MUST** be protected with either the same or newer packet protection keys than packets with lower packet numbers. An endpoint that successfully removes protection with old keys when newer keys were used for packets with lower packet numbers **MUST** treat this as a connection error of type KEY_UPDATE_ERROR.

6.5. Receiving with Different Keys

For receiving packets during a key update, packets protected with older keys might arrive if they were delayed by the network. Retaining old packet protection keys allows these packets to be successfully processed.

As packets protected with keys from the next key phase use the same Key Phase value as those protected with keys from the previous key phase, it is necessary to distinguish between the two if packets protected with old keys are to be processed. This can be done using packet numbers. A recovered packet number that is lower than any packet number from the current key phase uses the previous packet protection keys; a recovered packet number that is higher than any packet number from the current key phase requires the use of the next packet protection keys.

Some care is necessary to ensure that any process for selecting between previous, current, and next packet protection keys does not expose a timing side channel that might reveal which keys were used to remove packet protection. See Section 9.5 for more information.

Alternatively, endpoints can retain only two sets of packet protection keys, swapping previous for next after enough time has passed to allow for reordering in the network. In this case, the Key Phase bit alone can be used to select keys.

An endpoint **MAY** allow a period of approximately the Probe Timeout (PTO; see [QUIC-RECOVERY]) after promoting the next set of receive keys to be current before it creates the subsequent set of packet protection keys. These updated keys **MAY** replace the previous keys at that time. With the caveat that PTO is a subjective measure -- that is, a peer could have a different view of the RTT -- this time is expected to be long enough that any reordered packets would be declared lost by a peer even if they were acknowledged and short enough to allow a peer to initiate further key updates.

Endpoints need to allow for the possibility that a peer might not be able to decrypt packets that initiate a key update during the period when the peer retains old keys. Endpoints **SHOULD** wait three times the PTO before initiating a key update after receiving an acknowledgment that confirms that the previous key update was received. Failing to allow sufficient time could lead to packets being discarded.

An endpoint **SHOULD** retain old read keys for no more than three times the PTO after having received a packet protected using the new keys. After this period, old read keys and their corresponding secrets **SHOULD** be discarded.

6.6. Limits on AEAD Usage

This document sets usage limits for AEAD algorithms to ensure that overuse does not give an adversary a disproportionate advantage in attacking the confidentiality and integrity of communications when using QUIC.

The usage limits defined in TLS 1.3 exist for protection against attacks on confidentiality and apply to successful applications of AEAD protection. The integrity protections in authenticated encryption also depend on limiting the number of attempts to forge packets. TLS achieves this by closing connections after any record fails an authentication check. In comparison, QUIC ignores any packet that cannot be authenticated, allowing multiple forgery attempts.

QUIC accounts for AEAD confidentiality and integrity limits separately. The confidentiality limit applies to the number of packets encrypted with a given key. The integrity limit applies to the number of packets decrypted within a given connection. Details on enforcing these limits for each AEAD algorithm follow below.

Endpoints **MUST** count the number of encrypted packets for each set of keys. If the total number of encrypted packets with the same key exceeds the confidentiality limit for the selected AEAD, the endpoint **MUST** stop using those keys. Endpoints **MUST** initiate a key update before sending more protected packets than the confidentiality limit for the selected AEAD permits. If a key update is not possible or integrity limits are reached, the endpoint **MUST** stop using the connection and only send stateless resets in response to receiving packets. It is **RECOMMENDED** that endpoints immediately close the connection with a connection error of type AEAD_LIMIT_REACHED before reaching a state where key updates are not possible.

For AEAD_AES_128_GCM and AEAD_AES_256_GCM, the confidentiality limit is 2^{23} encrypted packets; see Appendix B.1. For AEAD_CHACHA20_POLY1305, the confidentiality limit is greater than the number of possible packets (2^{62}) and so can be disregarded. For AEAD_AES_128_CCM, the confidentiality limit is $2^{21.5}$ encrypted packets; see Appendix B.2. Applying a limit reduces the probability that an attacker can distinguish the AEAD in use from a random permutation; see [AEBounds], [ROBUST], and [GCM-MU].

In addition to counting packets sent, endpoints **MUST** count the number of received packets that fail authentication during the lifetime of a connection. If the total number of received packets that fail authentication within the connection, across all keys, exceeds the integrity limit for the selected AEAD, the endpoint **MUST** immediately close the connection with a connection error of type AEAD_LIMIT_REACHED and not process any more packets.

For AEAD_AES_128_GCM and AEAD_AES_256_GCM, the integrity limit is 2^{52} invalid packets; see Appendix B.1. For AEAD_CHACHA20_POLY1305, the integrity limit is 2^{36} invalid packets; see [AEBounds]. For AEAD_AES_128_CCM, the integrity limit is $2^{21.5}$ invalid packets; see Appendix B.2. Applying this limit reduces the probability that an attacker can successfully forge a packet; see [AEBounds], [ROBUST], and [GCM-MU].

Endpoints that limit the size of packets **MAY** use higher confidentiality and integrity limits; see Appendix B for details.

Future analyses and specifications **MAY** relax confidentiality or integrity limits for an AEAD.

Any TLS cipher suite that is specified for use with QUIC **MUST** define limits on the use of the associated AEAD function that preserves margins for confidentiality and integrity. That is, limits **MUST** be specified for the number of packets that can be authenticated and for the number of packets that can fail authentication. Providing a reference to any analysis upon which values are based -- and any assumptions used in that analysis -- allows limits to be adapted to varying usage conditions.

6.7. Key Update Error Code

The KEY_UPDATE_ERROR error code (0x0e) is used to signal errors related to key updates.

7. Security of Initial Messages

Initial packets are not protected with a secret key, so they are subject to potential tampering by an attacker. QUIC provides protection against attackers that cannot read packets but does not attempt to provide additional protection against attacks where the attacker can observe and inject packets. Some forms of tampering -- such as modifying the TLS messages themselves -- are detectable, but some -- such as modifying ACKs -- are not.

For example, an attacker could inject a packet containing an ACK frame to make it appear that a packet had not been received or to create a false impression of the state of the connection (e.g., by modifying the ACK Delay). Note that such a packet could cause a legitimate packet to be dropped as a duplicate. Implementations **SHOULD** use caution in relying on any data that is contained in Initial packets that is not otherwise authenticated.

It is also possible for the attacker to tamper with data that is carried in Handshake packets, but because that sort of tampering requires modifying TLS handshake messages, any such tampering will cause the TLS handshake to fail.

8. QUIC-Specific Adjustments to the TLS Handshake

Certain aspects of the TLS handshake are different when used with QUIC.

QUIC also requires additional features from TLS. In addition to negotiation of cryptographic parameters, the TLS handshake carries and authenticates values for QUIC transport parameters.

8.1. Protocol Negotiation

QUIC requires that the cryptographic handshake provide authenticated protocol negotiation. TLS uses Application-Layer Protocol Negotiation [ALPN] to select an application protocol. Unless another mechanism is used for agreeing on an application protocol, endpoints **MUST** use ALPN for this purpose.

When using ALPN, endpoints **MUST** immediately close a connection (see Section 10.2 of [QUIC-TRANSPORT]) with a no_application_protocol TLS alert (QUIC error code 0x0178; see Section 4.8) if an application protocol is not negotiated. While [ALPN] only specifies that servers use this alert, QUIC clients **MUST** use error 0x0178 to terminate a connection when ALPN negotiation fails.

An application protocol **MAY** restrict the QUIC versions that it can operate over. Servers **MUST** select an application protocol compatible with the QUIC version that the client has selected. The server **MUST** treat the inability to select a compatible application protocol as a connection error of type 0x0178 (no_application_protocol). Similarly, a client **MUST** treat the selection of an incompatible application protocol by a server as a connection error of type 0x0178.

8.2. QUIC Transport Parameters Extension

QUIC transport parameters are carried in a TLS extension. Different versions of QUIC might define a different method for negotiating transport configuration.

Including transport parameters in the TLS handshake provides integrity protection for these values.

```
enum {
   quic_transport_parameters(0x39), (65535)
} ExtensionType;
```

The extension_data field of the quic_transport_parameters extension contains a value that is defined by the version of QUIC that is in use.

The quic_transport_parameters extension is carried in the ClientHello and the EncryptedExtensions messages during the handshake. Endpoints **MUST** send the quic_transport_parameters extension; endpoints that receive ClientHello or EncryptedExtensions messages without the quic_transport_parameters extension **MUST** close the connection with an error of type 0x016d (equivalent to a fatal TLS missing_extension alert, see Section 4.8).

Transport parameters become available prior to the completion of the handshake. A server might use these values earlier than handshake completion. However, the value of transport parameters is not authenticated until the handshake completes, so any use of these parameters cannot depend on their authenticity. Any tampering with transport parameters will cause the handshake to fail.

Endpoints **MUST NOT** send this extension in a TLS connection that does not use QUIC (such as the use of TLS with TCP defined in [TLS13]). A fatal unsupported_extension alert **MUST** be sent by an implementation that supports this extension if the extension is received when the transport is not QUIC.

Negotiating the quic_transport_parameters extension causes the EndOfEarlyData to be removed; see Section 8.3.

8.3. Removing the EndOfEarlyData Message

The TLS EndOfEarlyData message is not used with QUIC. QUIC does not rely on this message to mark the end of 0-RTT data or to signal the change to Handshake keys.

Clients **MUST NOT** send the EndOfEarlyData message. A server **MUST** treat receipt of a CRYPTO frame in a 0-RTT packet as a connection error of type PROTOCOL_VIOLATION.

As a result, EndOfEarlyData does not appear in the TLS handshake transcript.

8.4. Prohibit TLS Middlebox Compatibility Mode

Appendix D.4 of [TLS13] describes an alteration to the TLS 1.3 handshake as a workaround for bugs in some middleboxes. The TLS 1.3 middlebox compatibility mode involves setting the legacy_session_id field to a 32-byte value in the ClientHello and ServerHello, then sending a change_cipher_spec record. Both field and record carry no semantic content and are ignored.

This mode has no use in QUIC as it only applies to middleboxes that interfere with TLS over TCP. QUIC also provides no means to carry a change_cipher_spec record. A client **MUST NOT** request the use of the TLS 1.3 compatibility mode. A server **SHOULD** treat the receipt of a TLS ClientHello with a non-empty legacy_session_id field as a connection error of type PROTOCOL_VIOLATION.

9. Security Considerations

All of the security considerations that apply to TLS also apply to the use of TLS in QUIC. Reading all of [TLS13] and its appendices is the best way to gain an understanding of the security properties of QUIC.

This section summarizes some of the more important security aspects specific to the TLS integration, though there are many security-relevant details in the remainder of the document.

9.1. Session Linkability

Use of TLS session tickets allows servers and possibly other entities to correlate connections made by the same client; see Section 4.5 for details.

9.2. Replay Attacks with 0-RTT

As described in Section 8 of [TLS13], use of TLS early data comes with an exposure to replay attack. The use of 0-RTT in QUIC is similarly vulnerable to replay attack.

Endpoints **MUST** implement and use the replay protections described in [TLS13], however it is recognized that these protections are imperfect. Therefore, additional consideration of the risk of replay is needed.

QUIC is not vulnerable to replay attack, except via the application protocol information it might carry. The management of QUIC protocol state based on the frame types defined in [QUIC-TRANSPORT] is not vulnerable to replay. Processing of QUIC frames is idempotent and cannot result in invalid connection states if frames are replayed, reordered, or lost. QUIC connections do not produce effects that last beyond the lifetime of the connection, except for those produced by the application protocol that QUIC serves.

TLS session tickets and address validation tokens are used to carry QUIC configuration information between connections, specifically, to enable a server to efficiently recover state that is used in connection establishment and address validation. These **MUST NOT** be used to communicate application semantics between endpoints; clients **MUST** treat them as opaque values. The potential for reuse of these tokens means that they require stronger protections against replay.

A server that accepts 0-RTT on a connection incurs a higher cost than accepting a connection without 0-RTT. This includes higher processing and computation costs. Servers need to consider the probability of replay and all associated costs when accepting 0-RTT.

Ultimately, the responsibility for managing the risks of replay attacks with 0-RTT lies with an application protocol. An application protocol that uses QUIC **MUST** describe how the protocol uses 0-RTT and the measures that are employed to protect against replay attack. An analysis of replay risk needs to consider all QUIC protocol features that carry application semantics.

Disabling 0-RTT entirely is the most effective defense against replay attack.

QUIC extensions **MUST** either describe how replay attacks affect their operation or prohibit the use of the extension in 0-RTT. Application protocols **MUST** either prohibit the use of extensions that carry application semantics in 0-RTT or provide replay mitigation strategies.

9.3. Packet Reflection Attack Mitigation

A small ClientHello that results in a large block of handshake messages from a server can be used in packet reflection attacks to amplify the traffic generated by an attacker.

QUIC includes three defenses against this attack. First, the packet containing a ClientHello **MUST** be padded to a minimum size. Second, if responding to an unverified source address, the server is forbidden to send more than three times as many bytes as the number of bytes it has received (see Section 8.1 of [QUIC-TRANSPORT]). Finally, because acknowledgments of Handshake packets are authenticated, a blind attacker cannot forge them. Put together, these defenses limit the level of amplification.

9.4. Header Protection Analysis

[NAN] analyzes authenticated encryption algorithms that provide nonce privacy, referred to as "Hide Nonce" (HN) transforms. The general header protection construction in this document is one of those algorithms (HN1). Header protection is applied after the packet protection AEAD, sampling a set of bytes (`sample`) from the AEAD output and encrypting the header field using a pseudorandom function (PRF) as follows:

```
protected_field = field XOR PRF(hp_key, sample)
```

The header protection variants in this document use a pseudorandom permutation (PRP) in place of a generic PRF. However, since all PRPs are also PRFs [IMC], these variants do not deviate from the HN1 construction.

As `hp_key` is distinct from the packet protection key, it follows that header protection achieves AE2 security as defined in [NAN] and therefore guarantees privacy of `field`, the protected packet header. Future header protection variants based on this construction **MUST** use a PRF to ensure equivalent security guarantees.

Use of the same key and ciphertext sample more than once risks compromising header protection. Protecting two different headers with the same key and ciphertext sample reveals the exclusive OR of the protected fields. Assuming that the AEAD acts as a PRF, if L bits are sampled, the odds of two ciphertext samples being identical approach $2^{-L/2}$, that is, the birthday bound. For the algorithms described in this document, that probability is one in 2^{64}.

To prevent an attacker from modifying packet headers, the header is transitively authenticated using packet protection; the entire packet header is part of the authenticated additional data. Protected fields that are falsified or modified can only be detected once the packet protection is removed.

9.5. Header Protection Timing Side Channels

An attacker could guess values for packet numbers or Key Phase and have an endpoint confirm guesses through timing side channels. Similarly, guesses for the packet number length can be tried and exposed. If the recipient of a packet discards packets with duplicate packet numbers without attempting to remove packet protection, they could reveal through timing side channels that the packet number matches a received packet. For authentication to be free from side channels, the entire process of header protection removal, packet number recovery, and packet protection removal **MUST** be applied together without timing and other side channels.

For the sending of packets, construction and protection of packet payloads and packet numbers **MUST** be free from side channels that would reveal the packet number or its encoded size.

During a key update, the time taken to generate new keys could reveal through timing side channels that a key update has occurred. Alternatively, where an attacker injects packets, this side channel could reveal the value of the Key Phase on injected packets. After receiving a key update, an endpoint **SHOULD** generate and save the next set of receive packet protection keys, as described in Section 6.3. By generating new keys before a key update is received, receipt of packets will not create timing signals that leak the value of the Key Phase.

This depends on not doing this key generation during packet processing, and it can require that endpoints maintain three sets of packet protection keys for receiving: for the previous key phase, for the current key phase, and for the next key phase. Endpoints can instead choose to defer generation of the next receive packet protection keys until they discard old keys so that only two sets of receive keys need to be retained at any point in time.

9.6. Key Diversity

In using TLS, the central key schedule of TLS is used. As a result of the TLS handshake messages being integrated into the calculation of secrets, the inclusion of the QUIC transport parameters extension ensures that the handshake and 1-RTT keys are not the same as those that might be produced by a server running TLS over TCP. To avoid the possibility of cross-protocol key synchronization, additional measures are provided to improve key separation.

The QUIC packet protection keys and IVs are derived using a different label than the equivalent keys in TLS.

To preserve this separation, a new version of QUIC **SHOULD** define new labels for key derivation for packet protection key and IV, plus the header protection keys. This version of QUIC uses the string "quic". Other versions can use a version-specific label in place of that string.

The initial secrets use a key that is specific to the negotiated QUIC version. New QUIC versions **SHOULD** define a new salt value used in calculating initial secrets.

9.7. Randomness

QUIC depends on endpoints being able to generate secure random numbers, both directly for protocol values such as the connection ID, and transitively via TLS. See [RFC4086] for guidance on secure random number generation.

10. IANA Considerations

IANA has registered a codepoint of 57 (or 0x39) for the quic_transport_parameters extension (defined in Section 8.2) in the "TLS ExtensionType Values" registry [TLS-REGISTRIES].

The Recommended column for this extension is marked Yes. The TLS 1.3 Column includes CH (ClientHello) and EE (EncryptedExtensions).

Value	Extension Name	TLS 1.3	Recommended	Reference
57	quic_transport_parameters	CH, EE	Y	This document

Table 2: TLS ExtensionType Values Registry Entry

11. References

11.1. Normative References

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

[AES] "Advanced encryption standard (AES)", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.197, November 2001, <https://doi.org/10.6028/nist.fips.197>.

[ALPN] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[CHACHA] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <https://www.rfc-editor.org/info/rfc8439>.

[HKDF] 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>.

[QUIC-RECOVERY] Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", RFC 9002, DOI 10.17487/RFC9002, May 2021, <https://www.rfc-editor.org/info/rfc9002>.

[QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

[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>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[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>.

[SHA] Dang, Q., "Secure Hash Standard", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.180-4, July 2015, <https://doi.org/10.6028/nist.fips.180-4>.

[TLS-REGISTRIES] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <https://www.rfc-editor.org/info/rfc8447>.

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

11.2. Informative References

[AEBounds] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", 28 August 2017, <https://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>.

[ASCII] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <https://www.rfc-editor.org/info/rfc20>.

[CCM-ANALYSIS] Jonsson, J., "On the Security of CTR + CBC-MAC", Selected Areas in Cryptography, SAC 2002, Lecture Notes in Computer Science, vol 2595, pp. 76-93, DOI 10.1007/3-540-36492-7_7, 2003, <https://doi.org/10.1007/3-540-36492-7_7>.

[COMPRESS] Ghedini, A. and V. Vasiliev, "TLS Certificate Compression", RFC 8879, DOI 10.17487/RFC8879, December 2020, <https://www.rfc-editor.org/info/rfc8879>.

[GCM-MU] Hoang, V., Tessaro, S., and A. Thiruvengadam, "The Multi-user Security of GCM, Revisited: Tight Bounds for Nonce Randomization", CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 1429-1440, DOI 10.1145/3243734.3243816, 2018, <https://doi.org/10.1145/3243734.3243816>.

[HTTP-REPLAY] Thomson, M., Nottingham, M., and W. Tarreau, "Using Early Data in HTTP", RFC 8470, DOI 10.17487/RFC8470, September 2018, <https://www.rfc-editor.org/info/rfc8470>.

[HTTP2-TLS13] Benjamin, D., "Using TLS 1.3 with HTTP/2", RFC 8740, DOI 10.17487/RFC8740, February 2020, <https://www.rfc-editor.org/info/rfc8740>.

[IMC] Katz, J. and Y. Lindell, "Introduction to Modern Cryptography, Second Edition", ISBN 978-1466570269, 6 November 2014.

[NAN] Bellare, M., Ng, R., and B. Tackmann, "Nonces Are Noticed: AEAD Revisited", Advances in Cryptology - CRYPTO 2019, Lecture Notes in Computer Science, vol 11692, pp. 235-265, DOI 10.1007/978-3-030-26948-7_9, 2019, <https://doi.org/10.1007/978-3-030-26948-7_9>.

[QUIC-HTTP] Bishop, M., Ed., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-34, 2 February 2021, <https://tools.ietf.org/html/draft-ietf-quic-http-34>.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <https://www.rfc-editor.org/info/rfc2818>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[ROBUST] Fischlin, M., Günther, F., and C. Janson, "Robust Channels: Handling Unreliable Networks in the Record Layers of QUIC and DTLS 1.3", 16 May 2020, <https://eprint.iacr.org/2020/718>.

Appendix A. Sample Packet Protection

This section shows examples of packet protection so that implementations can be verified incrementally. Samples of Initial packets from both client and server plus a Retry packet are defined. These packets use an 8-byte client-chosen Destination Connection ID of 0x8394c8f03e515708. Some intermediate values are included. All values are shown in hexadecimal.

A.1. Keys

The labels generated during the execution of the HKDF-Expand-Label function (that is, HkdfLabel.label) and part of the value given to the HKDF-Expand function in order to produce its output are:

client in: 00200f746c73313320636c69656e7420696e00

server in: 00200f746c7331332073657276657220696e00

quic key: 00100e746c7331332071756963206b657900

quic iv: 000c0d746c7331332071756963320697600

quic hp: 00100d746c7331332071756963320687000

The initial secret is common:

```
initial_secret = HKDF-Extract(initial_salt, cid)
    = 7db5df06e7a69e432496adedb0085192
      3595221596ae2ae9fb8115c1e9ed0a44
```

The secrets for protecting client packets are:

```
client_initial_secret
    = HKDF-Expand-Label(initial_secret, "client in", "", 32)
    = c00cf151ca5be075ed0ebfb5c80323c4
      2d6b7db67881289af4008f1f6c357aea

key = HKDF-Expand-Label(client_initial_secret, "quic key", "", 16)
    = 1f369613dd76d5467730efcbe3b1a22d

iv  = HKDF-Expand-Label(client_initial_secret, "quic iv", "", 12)
    = fa044b2f42a3fd3b46fb255c

hp  = HKDF-Expand-Label(client_initial_secret, "quic hp", "", 16)
    = 9f50449e04a0e810283a1e9933adedd2
```

The secrets for protecting server packets are:

```
server_initial_secret
    = HKDF-Expand-Label(initial_secret, "server in", "", 32)
    = 3c199828fd139efd216c155ad844cc81
      fb82fa8d7446fa7d78be803acdda951b

key = HKDF-Expand-Label(server_initial_secret, "quic key", "", 16)
    = cf3a5331653c364c88f0f379b6067e37

iv  = HKDF-Expand-Label(server_initial_secret, "quic iv", "", 12)
    = 0ac1493ca1905853b0bba03e

hp  = HKDF-Expand-Label(server_initial_secret, "quic hp", "", 16)
    = c206b8d9b9f0f37644430b490eeaa314
```

A.2. Client Initial

The client sends an Initial packet. The unprotected payload of this packet contains the following CRYPTO frame, plus enough PADDING frames to make a 1162-byte payload:

```
060040f1010000ed0303ebf8fa56f129 39b9584a3896472ec40bb863cfd3e868
04fe3a47f06a2b69484c000004130113 02010000c000000010000e00000b6578
616d706c652e636f6dff01000100000a 00080006001d0017001800100070005
04616c706e0005000501000000000033 00260024001d00209370b2c9caa47fba
baf4559fedba753de171fa71f50f1ce1 5d43e994ec74d748002b000302030400
0d0010000e0403050306030203080408 050806002d00020101001c0002400100
3900320408ffffffffffffffff050480 00ffff07048000ffff08011001048000
75300901100f088394c8f03e51570806 048000ffff
```

The unprotected header indicates a length of 1182 bytes: the 4-byte packet number, 1162 bytes of frames, and the 16-byte authentication tag. The header includes the connection ID and a packet number of 2:

```
c300000001088394c8f03e5157080000449e00000002
```

Protecting the payload produces output that is sampled for header protection. Because the header uses a 4-byte packet number encoding, the first 16 bytes of the protected payload is sampled and then applied to the header as follows:

```
sample = d1b1c98dd7689fb8ec11d242b123dc9b

mask = AES-ECB(hp, sample)[0..4]
     = 437b9aec36

header[0] ^= mask[0] & 0x0f
     = c0
header[18..21] ^= mask[1..4]
     = 7b9aec34
header = c000000001088394c8f03e5157080000449e7b9aec34
```

The resulting protected packet is:

```
c000000001088394c8f03e5157080000 449e7b9aec34d1b1c98dd7689fb8ec11
d242b123dc9bd8bab936b47d92ec356c 0bab7df5976d27cd449f63300099f399
1c260ec4c60d17b31f8429157bb35a12 82a643a8d2262cad67500cadb8e7378c
8eb7539ec4d4905fed1bee1fc8aafba1 7c750e2c7ace01e6005f80fcb7df6212
30c83711b39343fa028cea7f7fb5ff89 eac2308249a02252155e2347b63d58c5
457afd84d05dfffdb20392844ae81215 4682e9cf012f9021a6f0be17ddd0c208
4dce25ff9b06cde535d0f920a2db1bf3 62c23e596d11a4f5a6cf3948838a3aec
4e15daf8500a6ef69ec4e3feb6b1d98e 610ac8b7ec3faf6ad760b7bad1db4ba3
485e8a94dc250ae3fdb41ed15fb6a8e5 eba0fc3dd60bc8e30c5c4287e53805db
059ae0648db2f64264ed5e39be2e20d8 2df566da8dd5998ccabdae053060ae6c
7b4378e846d29f37ed7b4ea9ec5d82e7 961b7f25a9323851f681d582363aa5f8
9937f5a67258bf63ad6f1a0b1d96dbd4 faddfcefc5266ba6611722395c906556
be52afe3f565636ad1b17d508b73d874 3eeb524be22b3dcbc2c7468d54119c74
68449a13d8e3b95811a198f3491de3e7 fe942b330407abf82a4ed7c1b311663a
c69890f4157015853d91e923037c227a 33cdd5ec281ca3f79c44546b9d90ca00
f064c99e3dd97911d39fe9c5d0b23a22 9a234cb36186c4819e8b9c5927726632
291d6a418211cc2962e20fe47feb3edf 330f2c603a9d48c0fcb5699dbfe58964
25c5bac4aee82e57a85aaf4e2513e4f0 5796b07ba2ee47d80506f8d2c25e50fd
14de71e6c418559302f939b0e1abd576 f279c4b2e0feb85c1f28ff18f58891ff
ef132eef2fa09346aee33c28eb130ff2 8f5b766953334113211996d20011a198
e3fc433f9f2541010ae17c1bf202580f 6047472fb36857fe843b19f5984009dd
c324044e847a4f4a0ab34f719595de37 252d6235365e9b84392b061085349d73
203a4a13e96f5432ec0fd4a1ee65accd d5e3904df54c1da510b0ff20dcc0c77f
cb2c0e0eb605cb0504db87632cf3d8b4 dae6e705769d1de354270123cb11450e
fc60ac47683d7b8d0f811365565fd98c 4c8eb936bcab8d069fc33bd801b03ade
a2e1fbc5aa463d08ca19896d2bf59a07 1b851e6c239052172f296bfb5e724047
90a2181014f3b94a4e97d117b4381303 68cc39dbb2d198065ae3986547926cd2
162f40a29f0c3c8745c0f50fba3852e5 66d44575c29d39a03f0cda721984b6f4
40591f355e12d439ff150aab7613499d bd49adabc8676eef023b15b65bfc5ca0
6948109f23f350db82123535eb8a7433 bdabcb909271a6ecbcb58b936a88cd4e
8f2e6ff5800175f113253d8fa9ca8885 c2f552e657dc603f252e1a8e308f76f0
be79e2fb8f5d5fbbe2e30ecadd220723 c8c0aea8078cdfcb3868263ff8f09400
54da48781893a7e49ad5aff4af300cd8 04a6b6279ab3ff3afb64491c85194aab
760d58a606654f9f4400e8b38591356f bf6425aca26dc85244259ff2b19c41b9
f96f3ca9ec1dde434da7d2d392b905dd f3d1f9af93d1af5950bd493f5aa731b4
056df31bd267b6b90a079831aaf579be 0a39013137aac6d404f518cfd4684064
7e78bfe706ca4cf5e9c5453e9f7cfd2b 8b4c8d169a44e55c88d4a9a7f9474241
e221af44860018ab0856972e194cd934
```

A.3. Server Initial

The server sends the following payload in response, including an ACK frame, a CRYPTO frame, and no PADDING frames:

```
0200000000600405a020000560303ee fce7f7b37ba1d1632e96677825ddf739
88cfc79825df566dc5430b9a045a1200 130100002e00330024001d00209d3c94
0d89690b84d08a60993c144eca684d10 81287c834d5311bcf32bb9da1a002b00
020304
```

The header from the server includes a new connection ID and a 2-byte packet number encoding for a packet number of 1:

```
c1000000010008f067a5502a4262b50040750001
```

As a result, after protection, the header protection sample is taken starting from the third protected byte:

```
sample = 2cd0991cd25b0aac406a5816b6394100
mask   = 2ec0d8356a
header = cf000000010008f067a5502a4262b5004075c0d9
```

The final protected packet is then:

```
cf000000010008f067a5502a4262b500 4075c0d95a482cd0991cd25b0aac406a
5816b6394100f37a1c69797554780bb3 8cc5a99f5ede4cf73c3ec2493a1839b3
dbcba3f6ea46c5b7684df3548e7ddeb9 c3bf9c73cc3f3bded74b562bfb19fb84
022f8ef4cdd93795d77d06edbb7aaf2f 58891850abbdca3d20398c276456cbc4
2158407dd074ee
```

A.4. Retry

This shows a Retry packet that might be sent in response to the Initial packet in Appendix A.2. The integrity check includes the client-chosen connection ID value of 0x8394c8f03e515708, but that value is not included in the final Retry packet:

```
ff000000010008f067a5502a4262b574 6f6b656e04a265ba2eff4d829058fb3f
0f2496ba
```

A.5. ChaCha20-Poly1305 Short Header Packet

This example shows some of the steps required to protect a packet with a short header. This example uses AEAD_CHACHA20_POLY1305.

In this example, TLS produces an application write secret from which a server uses HKDF-Expand-Label to produce four values: a key, an IV, a header protection key, and the secret that will be used after keys are updated (this last value is not used further in this example).

```
secret
    = 9ac312a7f877468ebe69422748ad00a1
      5443f18203a07d6060f688f30f21632b

key = HKDF-Expand-Label(secret, "quic key", "", 32)
    = c6d98ff3441c3fe1b2182094f69caa2e
      d4b716b65488960a7a984979fb23e1c8

iv  = HKDF-Expand-Label(secret, "quic iv", "", 12)
    = e0459b3474bdd0e44a41c144

hp  = HKDF-Expand-Label(secret, "quic hp", "", 32)
    = 25a282b9e82f06f21f488917a4fc8f1b
      73573685608597d0efcb076b0ab7a7a4

ku  = HKDF-Expand-Label(secret, "quic ku", "", 32)
    = 1223504755036d556342ee9361d25342
      1a826c9ecdf3c7148684b36b714881f9
```

The following shows the steps involved in protecting a minimal packet with an empty Destination Connection ID. This packet contains a single PING frame (that is, a payload of just 0x01) and has a packet number of 654360564. In this example, using a packet number of length 3 (that is, 49140 is encoded) avoids having to pad the payload of the packet; PADDING frames would be needed if the packet number is encoded on fewer bytes.

```
pn                 = 654360564 (decimal)
nonce              = e0459b3474bdd0e46d417eb0
unprotected header = 4200bff4
payload plaintext  = 01
payload ciphertext = 655e5cd55c41f69080575d7999c25a5bfb
```

The resulting ciphertext is the minimum size possible. One byte is skipped to produce the sample for header protection.

```
sample = 5e5cd55c41f69080575d7999c25a5bfb
mask   = aefefe7d03
header = 4cfe4189
```

The protected packet is the smallest possible packet size of 21 bytes.

```
packet = 4cfe4189655e5cd55c41f69080575d7999c25a5bfb
```

Appendix B. AEAD Algorithm Analysis

This section documents analyses used in deriving AEAD algorithm limits for AEAD_AES_128_GCM, AEAD_AES_128_CCM, and AEAD_AES_256_GCM. The analyses that follow use symbols for multiplication (*), division (/), and exponentiation (^), plus parentheses for establishing precedence. The following symbols are also used:

t: The size of the authentication tag in bits. For these ciphers, t is 128.

n: The size of the block function in bits. For these ciphers, n is 128.

k: The size of the key in bits. This is 128 for AEAD_AES_128_GCM and AEAD_AES_128_CCM; 256 for AEAD_AES_256_GCM.

l: The number of blocks in each packet (see below).

q: The number of genuine packets created and protected by endpoints. This value is the bound on the number of packets that can be protected before updating keys.

v: The number of forged packets that endpoints will accept. This value is the bound on the number of forged packets that an endpoint can reject before updating keys.

o: The amount of offline ideal cipher queries made by an adversary.

The analyses that follow rely on a count of the number of block operations involved in producing each message. This analysis is performed for packets of size up to 2^{11} ($l = 2^7$) and 2^{16} ($l = 2^{12}$). A size of 2^{11} is expected to be a limit that matches common deployment patterns, whereas the 2^{16} is the maximum possible size of a QUIC packet. Only endpoints that strictly limit packet size can use the larger confidentiality and integrity limits that are derived using the smaller packet size.

For AEAD_AES_128_GCM and AEAD_AES_256_GCM, the message length (l) is the length of the associated data in blocks plus the length of the plaintext in blocks.

For AEAD_AES_128_CCM, the total number of block cipher operations is the sum of the following: the length of the associated data in blocks, the length of the ciphertext in blocks, the length of the plaintext in blocks, plus 1. In this analysis, this is simplified to a value of twice the length of the packet in blocks (that is, $2l = 2^8$ for packets that are limited to 2^{11} bytes, or $2l = 2^{13}$ otherwise). This simplification is based on the packet containing all of the associated data and ciphertext. This results in a one to three block overestimation of the number of operations per packet.

B.1. Analysis of AEAD_AES_128_GCM and AEAD_AES_256_GCM Usage Limits

[GCM-MU] specifies concrete bounds for AEAD_AES_128_GCM and AEAD_AES_256_GCM as used in TLS 1.3 and QUIC. This section documents this analysis using several simplifying assumptions:

- The number of ciphertext blocks an attacker uses in forgery attempts is bounded by v * l, which is the number of forgery attempts multiplied by the size of each packet (in blocks).
- The amount of offline work done by an attacker does not dominate other factors in the analysis.

The bounds in [GCM-MU] are tighter and more complete than those used in [AEBounds], which allows for larger limits than those described in [TLS13].

B.1.1. Confidentiality Limit

For confidentiality, Theorem (4.3) in [GCM-MU] establishes that, for a single user that does not repeat nonces, the dominant term in determining the distinguishing advantage between a real and random AEAD algorithm gained by an attacker is:

```
2 * (q * l)^2 / 2^n
```

For a target advantage of 2^{-57}, this results in the relation:

```
q <= 2^35 / l
```

Thus, endpoints that do not send packets larger than 2^{11} bytes cannot protect more than 2^{28} packets in a single connection without causing an attacker to gain a more significant advantage than the target of 2^{-57}. The limit for endpoints that allow for the packet size to be as large as 2^{16} is instead 2^{23}.

B.1.2. Integrity Limit

For integrity, Theorem (4.3) in [GCM-MU] establishes that an attacker gains an advantage in successfully forging a packet of no more than the following:

```
(1 / 2^(8 * n)) + ((2 * v) / 2^(2 * n))
        + ((2 * o * v) / 2^(k + n)) + (n * (v + (v * l)) / 2^k)
```

The goal is to limit this advantage to 2^{-57}. For AEAD_AES_128_GCM, the fourth term in this inequality dominates the rest, so the others can be removed without significant effect on the result. This produces the following approximation:

```
v <= 2^64 / l
```

Endpoints that do not attempt to remove protection from packets larger than 2^{11} bytes can attempt to remove protection from at most 2^{57} packets. Endpoints that do not restrict the size of processed packets can attempt to remove protection from at most 2^{52} packets.

For AEAD_AES_256_GCM, the same term dominates, but the larger value of k produces the following approximation:

```
v <= 2^192 / l
```

This is substantially larger than the limit for AEAD_AES_128_GCM. However, this document recommends that the same limit be applied to both functions as either limit is acceptably large.

B.2. Analysis of AEAD_AES_128_CCM Usage Limits

TLS [TLS13] and [AEBounds] do not specify limits on usage for AEAD_AES_128_CCM. However, any AEAD that is used with QUIC requires limits on use that ensure that both confidentiality and integrity are preserved. This section documents that analysis.

[CCM-ANALYSIS] is used as the basis of this analysis. The results of that analysis are used to derive usage limits that are based on those chosen in [TLS13].

For confidentiality, Theorem 2 in [CCM-ANALYSIS] establishes that an attacker gains a distinguishing advantage over an ideal pseudorandom permutation (PRP) of no more than the following:

```
(2l * q)^2 / 2^n
```

The integrity limit in Theorem 1 in [CCM-ANALYSIS] provides an attacker a strictly higher advantage for the same number of messages. As the targets for the confidentiality advantage and the integrity advantage are the same, only Theorem 1 needs to be considered.

Theorem 1 establishes that an attacker gains an advantage over an ideal PRP of no more than the following:

```
v / 2^t + (2l * (v + q))^2 / 2^n
```

As `t` and `n` are both 128, the first term is negligible relative to the second, so that term can be removed without a significant effect on the result.

This produces a relation that combines both encryption and decryption attempts with the same limit as that produced by the theorem for confidentiality alone. For a target advantage of 2^{-57}, this results in the following:

```
v + q <= 2^34.5 / l
```

By setting `q = v`, values for both confidentiality and integrity limits can be produced. Endpoints that limit packets to 2^{11} bytes therefore have both confidentiality and integrity limits of $2^{26.5}$ packets. Endpoints that do not restrict packet size have a limit of $2^{21.5}$.

Contributors

The IETF QUIC Working Group received an enormous amount of support from many people. The following people provided substantive contributions to this document:

- Adam Langley
- Alessandro Ghedini
- Christian Huitema
- Christopher Wood
- David Schinazi
- Dragana Damjanovic
- Eric Rescorla
- Felix Günther
- Ian Swett
- Jana Iyengar
- 奥 一穂 (Kazuho Oku)
- Marten Seemann
- Martin Duke
- Mike Bishop
- Mikkel Fahnøe Jørgensen
- Nick Banks
- Nick Harper
- Roberto Peon
- Rui Paulo
- Ryan Hamilton
- Victor Vasiliev

Authors' Addresses

Martin Thomson (EDITOR)
Mozilla
Email: mt@lowentropy.net

Sean Turner (EDITOR)
sn3rd
Email: sean@sn3rd.com