The ML-KEM Braid Protocol

Rolfe Schmidt

Revision 1 , 2025-02-21 Last updated: 2025-09-26

Contents

1. Introduction

The ML-KEM Braid is a *Sparse Continuous Key Agreement (SCKA)* protocol that uses NIST standardized ML-KEM [1] to allow two parties to produce a sequence of post-quantum secure shared secrets. These shared secrets have Forward Secrecy (FS) and Post-Compromise Security (PCS) properties that can carry over to higher level protocols such as a Double Ratchet protocol for secure messaging [2].

1.1 Sparse Continuous Key Agreement

The ping-pong style key exchange at the heart of the classical Double Ratchet protocol is called a *Continuous Key Agreement (CKA)* protocol. It has the desirable feature that it emits new shared secrets at every round trip, and these secrets can be passed to a higher-level protocol - like the Double Ratchet - to provide Post-Compromise Security.

The messages that must be passed for quantum-secure key agreement are much larger, though, so in the presence of bandwidth constraints a protocol may send these messages in pieces and will not be able to emit new shared secrets in each round trip. To capture this we follow [3] and use the notion of *Sparse* Continuous Key Agreement (SCKA).

An SCKA protocol outputs an ordered sequence of shared secrets, and the position of a shared secret in this sequence is an unsigned integer called the *epoch identifier*, or simply the *epoch*. The notion of *epoch* is implicit in the Diffie-Hellman ratchet of the classic Double Ratchet protocol [2], but in that context the public ratchet key is sufficient to serve as the epoch identifier. In contrast, we need stronger property, such as the ability to strictly order the epochs without gaps, in order to ensure that we use the output keys correctly.

In an SCKA protocol, each party maintains state and exposes two functions:

- ***Send(state) → (msg, sending_epoch, output_key)***: Updates the state and returns *msg*, a message to be processed by the other party, *sending_epoch*, the identifier of the latest epoch guaranteed to be known by the other party on receipt of *msg*, and *output_key*, a nullable pair containing an epoch identifier and a shared secret for that epoch.
- ***Receive(state, msg) → (receiving_epoch, output_key)***: Updates the state and returns *receiving_epoch*, the epoch identifier that was output by the other party as *sending_epoch* when they called *send()* to generate *msg*, and *output_key*, a nullable pair containing an epoch identifier and a shared secret for that epoch.

Precise correctness and security definitions are given in [3]. Informally, for correctness, the two parties in the protocol must agree on the sequence of keys emitted and the values of *sending_epoch* and *receiving_epoch* must indicate the most recent epochs that can be used correctly.

- **Session key consistency:** If Alice and Bob output keys *(ep, k)* and *(ep', k')* respectively where *ep = ep'*, then *k = k'*.
- **Per-participant epoch uniqueness:** Each party outputs at most one key per epoch.
- **Sender epoch knowledge:** When *Send()* returns sending_epoch, the sender possesses, or has possessed, the key for that epoch and all earlier epochs.
- **Receiver epoch knowledge:** When *Receive()* returns receiving_epoch, the receiver possesses, or has possessed, the key for that epoch and all earlier epochs.

- **Epoch agreement:** *sending_epoch* from *Send()* equals *receiving_epoch* from the corresponding *Receive()*. Specifically, if Alice (respectively, Bob) calls *Send()* which returns *msg* and sending_epoch, then when Bob (respectively, Alice) calls *Receive(msg)*, it must return *receiving_epoch = sending_epoch*.

1.2 Incremental KEMs

Some lattice-based Key Encapsulation Mechanisms (KEMs) based on the Learning With Errors assumption take the form of a "noisy key exchange" followed by a small reconciliation message that allows the parties to arrive at an exact shared secret. Because of this, the ciphertexts for these KEMs consist of two parts: *ct1* is a possibly compressed public key and *ct2* is a reconciliation message. The important observation is that *ct1* can be computed without complete knowledge of any encapsulation key.

To allow KEM users to take advantage of this fact, these KEMs can expose the following *incremental interface*:

- **KeyGen(randomness) → (dk, ek_header, ek_vector)**: Takes an array of random bits and returns a decapsulation key, *dk*, an *ek_header* with all information needed for a recipient to calculate *ct1*, and the "vector" part of the encapsulation key, *ek_vector*. We note that for some KEMs it is possible that the header could be empty.
- **Encaps1(ek_header, randomness) → (encaps_secret, ct1, shared_secret)**: Takes an encapsulation key header and an array of random bits as input and samples the first part of a new ciphertext. It returns *encaps_secret*, an encapsulation secret that holds the information needed to complete the encapsulation, *ct1*, the first component of a ciphertext, and *shared_secret*, the shared secret encapsulated by the ciphertext.
- **Encaps2(encaps_secret, ek_header, ek_vector) → ct2**: Takes an encapsulation secret, encapsulation key header, and encapsulation key vector and completes the encapsulation process, returning a reconciliation message, *ct2*.

- **Decaps(dk, ct1, ct2) → shared_secret**: Takes a decapsulation key and a complete ciphertext and returns the encapsulated shared secret.

1.2.1 ML-KEM as an Incremental KEM

ML-KEM [1] encapsulation keys consist of a 32-byte seed followed by a larger noisy vector. This seed is required to compute the "compressed public key" part of a ciphertext, *ct1*. Due to the Fujisaki-Okamoto transform [4] variant used by ML-KEM, we also need to know the SHA3-256 hash of the full encapsulation key to compute *ct1*. Thus an encapsulation key header for ML-KEM has the following fields:

- **ek_seed**: A 32-byte seed.
- **hek**: The SHA3-256 hash of *ek_seed* // *ek_vector*.

As discussed in Section 3.2, ML-KEM use of the encapsulation key hash gives it key binding properties that go beyond standard IND-CCA security and this should be considered when evaluating the use of any alternate KEM.

1.3 Chunking with Erasure Codes

An SCKA protocol sends large messages in pieces and must do this in a way that is robust, even in an adversarial network environment. To accomplish this a protocol can use *erasure codes* or *fountain codes*. Informally this can be thought of as breaking a message into a stream of chunks, and in this document any mention of a "chunk" of a message refers to a codeword of an erasure code.

2. The ML-KEM Braid Protocol

2.1 Overview

The ML-KEM Braid protocol takes advantage of the incremental interface ML-KEM described above to parallelize message sending and speed recovery from compromise. Specifically, the incremental interface allows *ct1* to be sampled after receiving just a *header*, after which *ct1* and *ek_vector* - the largest components of the ciphertext and encapsulation key - can be sent in parallel.

The following is a high level description of one epoch of the ML-KEM Braid protocol.

- A samples a new ML-KEM keypair: *(dk, ek_seed, ek_vector) = ML-KEM-KeyGen()*.
- A encodes a header message, *ek_seed* // *SHA3-256(ek_seed* // *ek_vector)*, and begins sending it to B in chunks.
- When B receives enough chunks to reconstruct the message, they decode and compute *(encaps_secret, ct1, shared_secret) = ML-KEM-Encaps1(ek_seed, SHA3-256(ek_seed* // *ek_vector))*. B stores *encaps_secret* and *shared_secret* for later use.

- B encodes *ct1* and begins sending it to A in chunks.
- When A receives the first chunk of *ct1*, they stop sending chunks of the header and start sending chunks of *ek_vector*.
- Now A and B send their messages in parallel.
- When A receives all of *ct1* they begin acknowledging the receipt in future messages sent to B.
- Once B receives all of *ek_vector* and receives an acknowledgment that *ct1* was received, they compute *ct2 = ML-KEM-Encaps2(encaps_secret, ek_seed, ek_vector)*.
- B encodes *ct2* and begins sending it to A in chunks.
- When A receives the first chunk of *ct2*, they stop sending chunks of *ek_vector*.
- When A receives all of *ct2*, they decapsulate the shared secret: *shared_secret = ML-KEM-Decaps(dk, ct1, ct2)*.
- Now A and B switch roles. A begins waiting for a header message from B, and indicates it has moved to the next epoch when sending messages to B.
- Once B receives a message showing that A has advanced to the next epoch, they sample a new keypair and begin again.

While this captures the main flow of the protocol, it does not tell us how A and B know *when* they can use the keys returned by the protocol. Clearly, when B returns *shared_secret* above, they cannot use it to encrypt messages to A because A does not know *shared_secret* yet. This will be addressed by the values *sending_epoch* and *receiving_epoch* returned from the functions defined below - a value that tells the caller what latest epoch key known by both parties at the time a message was created.

The protocol below also performs optional authentication, with details presented in Section 2.4 and discussed further in Section 3.3.

2.2 Parameters

- ***KEM***: An IND-CPA secure Key Encapsulation Mechanism that offers an incremental interface. For this document it will be one of *ML-KEM-512*, *ML-KEM-768*, or *ML-KEM-1024*. The ***KEM*** exposes the incremental interface described in Section 1.2
- **Constants**: Several constants are also associated with the KEM and are needed in the protocol description:

Constant	ML-KEM 512	ML-KEM 768	ML-KEM 1024
HEADER_SIZE	64	64	64
EK_SIZE	768	1152	1536
CT1_SIZE	640	960	1408
CT2_SIZE	128	128	160

- ***Encode/Decode***: An erasure code or fountain code that can encode a

long message into a stream of codewords, or chunks, so that when the receiver gets a sufficient number of these chunks, regardless of order or dropped codewords, they will be able to reconstruct the original message. Reed-Solomon based erasure codes over $GF(2^{16})^{w/2}$ for a chunk size of w bytes are recommended.

  - **Encode(byte_array) → encoder**: Returns a stateful encoding object that produces a stream of codewords, or *chunks*, that can be decoded to reconstruct *byte_array*. These codewords are accessed by calling the method *encoder.next_chunk()*.
  - **Decoder.new(message_size) → decoder**: Returns a stateful decoding object that will decode a message of length *message_size* from a set of codewords produced by a single encoder. It exposes the functions:
    * **decoder.add_chunk(chunk)**: Adds a codeword to the decoder's state.
    * **decoder.has_message() → bool**: Returns true when the decoder has received enough codewords to reconstruct the message.
    * **decoder.message() → maybe_byte_array**: Returns the reconstructed message if possible, otherwise returns Null.

- ***EPOCH_TYPE***: The unsigned integer type used to represent epochs. We recommend using unsigned 64-bit integers.
- ***ToBytes(epoch)***: Represent an epoch as a byte string. When *EPOCH_TYPE* is a 64-bit unsigned integer, use of big-endian encoding is recommended.
- ***MAC(mac_key, msg)***: A message authentication code. *HMAC-SHA256* is recommended.
- ***MAC_SIZE***: Size of *MAC*'s output, in bytes.
- ***PROTOCOL_INFO***: The concatenation of a protocol identifier, a string representation of *KEM*, and a string representation of *MAC*, separated with the delimiter "_", such as "`MyProtocol_MLKEM768_SHA-256`". The string representations of the ML-KEM Braid parameters are defined by the implementer.
- ***KDF_AUTH(root_key, update_key, epoch)***: 64 bytes of output from the HKDF algorithm [5] using *hash* with inputs:
  - *HKDF input key material* = *update_key*
  - *HKDF salt* = *root_key*
  - *HKDF info* = PROTOCOL_INFO || ":Authenticator Update" || ToBytes(epoch)
  - *HKDF length* = 64
- ***KDF_OK(shared_secret, epoch)***: 32 bytes of output from the HKDF algorithm [5] using *hash* with inputs:

- *HKDF input key material* = *shared_secret*
- *HKDF salt* = A zero-filled byte sequence with length equal to the *hash* output length, in bytes.
- *HKDF info* = PROTOCOL_INFO || ":SCKA Key" || ToBytes(epoch)
- *HKDF length* = 32

2.3 Messages

Messages consist of the following fields:

- **epoch** (unsigned integer): Current epoch being negotiated
- **type** (enum): One of *{None, Hdr, Ek, EkCt1Ack, Ct1Ack, Ct1, Ct2}* with the following meanings:
  - *None*: There is no payload
  - *Hdr*: The payload contains a *chunk* of the header.
  - *Ek*: The payload contains a *chunk* of the encapsulation key.
  - *EkCt1Ack*: The payload contains a *chunk* of the encapsulation key, and the sender has completely received *ct1*.
  - *Ct1Ack*: No payload, but the sender has completely received *ct1*.
  - *Ct1*: The payload contains a *chunk* of *ct1*.
  - *Ct2*: The payload contains a *chunk* of *ct2*.
- **data** (bytes, optional): Erasure code chunk when *type* is not one of *{ None, Ct1Ack }*

In what follows we will describe messages logically using object notation. Implementations may use a custom compact binary format or a general purpose serialization tool such as Protocol Buffers [6] to encode these messages. In the presence of bandwidth limits, implementers should consider that a custom format may allow larger chunk sizes and correspondingly improve post-compromise security (See Section 3.4).

2.4 Internal Authentication

While messaging protocols such as the Double Ratchet [2] provide ratcheted message authentication through the use of AEAD or explicit MACs on messages, it may be desirable for an SCKA protocol to provide internal authenticity guarantees. We attain this using a *Ratcheted Authenticator*.

Ratcheted Authenticator state variables

The Ratcheted Authenticator holds the following state:

- ***root_key***: a 32 byte value.
- ***mac_key***: a 32 byte key for use with *MAC*.

Ratcheted Authenticator functions

The Ratcheted Authenticator offers a function to update the internal state with new entropy as well as functions to compute and verify MACs on ciphertexts and header messages:

```
def Authenticator.Init(auth_state, epoch, key):
    auth_state = {root_key: '\0'*32, mac_key: None }
    auth_state.Update(epoch, key)

def Authenticator.Update(auth_state, epoch, key):
    auth_state.root_key, auth_state.mac_key
      = KDF_AUTH(auth_state.root_key, key, epoch)

def Authenticator.MacHdr(auth_state, epoch, hdr):
    return MAC(
        auth_state.mac_key,
        PROTOCOL_INFO || ":ekheader" || epoch || hdr,
        MAC_SIZE)

def Authenticator.MacCt(auth_state, epoch, ct):
    return MAC(
        auth_state.mac_key,
        PROTOCOL_INFO || ":ciphertext" || epoch || ct,
        MAC_SIZE)

def Authenticator.VfyHdr(auth_state, epoch, hdr, expected_mac):
    if expected_mac != auth_state.MacHdr(epoch, hdr):
        FAIL

def Authenticator.VfyCt(auth_state, epoch, ct, expected_mac):
    if expected_mac != auth_state.MacCt(epoch, ct):
        FAIL
```

In the event of a verification failure, protocol participants should not proceed with the ML-KEM Braid session and should negotiate a new ML-KEM Braid session.

2.5 State Machine and Transitions

We describe the protocol as a state machine that transitions from state to state when sending or receiving messages. The states and transitions can be seen in the following figure, which can serve as a helpful reference in the detailed descriptions that follow.

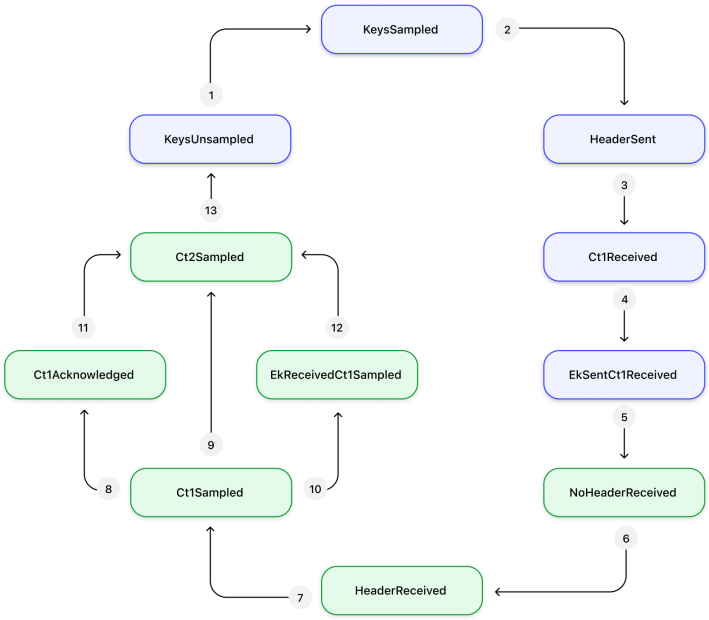

Figure 1: State machine transitions for the ML-KEM Braid Protocol. Each transition is labeled with a number that can be found in the pseudocode below.

All states of the agents contain at least the following two variables:

- *epoch*: an unsigned integer identifying the epoch of the key being negotiated.
- *auth*: an Authenticator object.

The following describes the state of an agent when they are transmitting an encapsulation key and awaiting the corresponding ciphertext. For each state we define the SCKA *Send()* and *Receive()* functions.

KeysUnsampled

Represents an agent that is ready to sample a new KEM keypair on the next *send* event. It carries no additional state.

When sending a message, the **KeysUnsampled** agent samples a new keypair, starts sending a header message, and transitions into the **KeysSampled** state. The **KeysUnsampled** agent ignores all messages it receives:

```
def KeysUnsampled.Send(state):
  # Generate keypair and header
  (dk, ek_seed, ek_vector) = KEM.KeyGen()
  hek = SHA3-256(ek_seed || ek_vector)
  header = ek_seed || hek
  mac = state.auth.MacHdr(state.epoch, header)
  header_encoder = Encode(header || mac)

  # Generate message
  chunk = header_encoder.next_chunk()
  msg = {epoch: state.epoch, type: Hdr, data: chunk}

  # Update state
  # Transition (1)
  state = KeysSampled(
    state.epoch,
    state.auth,
    dk,
    ek_seed,
    ek_vector,
    hek,
    header_encoder)

  # Return values
  output_key = None
  sending_epoch = state.epoch - 1
  return (msg, sending_epoch, output_key)

def KeysUnsampled.Receive(state, msg):
  # No action taken
```

```
  output_key = None
  receiving_epoch = state.epoch - 1
  return (receiving_epoch, output_key)
```

KeysSampled

Represents an agent that has sampled a KEM keypair and is sending the header. Additional state includes:

- *dk*: a KEM decapsulation key
- *ek_vector*: vector part of a KEM encapsulation key
- *header_encoder*

The **KeysSampled** agent sends chunks of the header. When it receives a message of type *Ct1* it knows that the other party has received the complete header so it transitions into the **HeaderSent** state, in which it will begin sending chunks of *ek_vector*:

```
def KeysSampled.Send(state):
  # Generate next header chunk
  chunk = state.header_encoder.next_chunk()
  msg = {epoch: state.epoch, type: Hdr, data: chunk}

  # Return values
  output_key = None
  sending_epoch = state.epoch - 1
  return (msg, sending_epoch, output_key)

def KeysSampled.Receive(state, msg):
  output_key = None
  receiving_epoch = state.epoch - 1

  if msg.epoch == state.epoch and msg.type == Ct1:
      # Initialize ct1 decoder and ek encoder
      ct1_decoder = Decoder.new(KEM.CT1_SIZE)
      ct1_decoder.add_chunk(msg.data)
      ek_encoder = Encode(state.ek_vector)

      # Update state
      # Transition (2)
      state = HeaderSent(
        state.epoch,
        state.auth,
        state.dk,
        ct1_decoder,
        ek_encoder)
```

```
  return (receiving_epoch, output_key)
```

HeaderSent

Represents an agent that has completed sending a header, is currently sending an *ek_vector*, and is receiving chunks of *ct1*. Additional state includes:

- *dk*: a KEM decapsulation key
- *ct1_decoder*
- *ek_encoder*

In the **HeaderSent** state, an agent sends chunks of its *ek_vector*. When receiving a message of type *Ct1* for the current epoch, if it has enough chunks to decode the incoming *ct1*, it transitions to the *Ct1Received* state:

```
def HeaderSent.Send(state):
  # Generate next ek_vector chunk
  chunk = state.ek_encoder.next_chunk()
  msg = {epoch: state.epoch, type: Ek, data: chunk}

  # Return values
  output_key = None
  sending_epoch = state.epoch - 1
  return (msg, sending_epoch, output_key)

def HeaderSent.Receive(state, msg):
  output_key = None
  receiving_epoch = state.epoch - 1

  if msg.epoch == state.epoch and msg.type == Ct1:
      # Add chunk to decoder
      state.ct1_decoder.add_chunk(msg.data)

      # Check if ct1 is complete
      if state.ct1_decoder.has_message():
          ct1 = state.ct1_decoder.message()

          # Update state
          # Transition (3)
          state = Ct1Received(
            state.epoch,
            state.auth,
            state.dk,
            ct1,
            state.ek_encoder)
```

```
  return (receiving_epoch, output_key)
```

Ct1Received

Represents an agent that has completely received *ct1* and is still sending chunks of *ek_vector*. Additional state includes:

- *dk*: a KEM decapsulation key
- *ct1*: The compressed public key part of a KEM ciphertext
- *ek_encoder*

In the **Ct1Received** state an agent sends chunks of the *ek_vector* until it receives a chunk of *ct2*. At that point it knows *ek_vector* has been received so it transitions into the **EkSentCt1Received** state:

```
def Ct1Received.Send(state):
  # Generate next ek_vector chunk with acknowledgment
  chunk = state.ek_encoder.next_chunk()
  msg = {epoch: state.epoch, type: EkCt1Ack, data: chunk}

  # Return values
  output_key = None
  sending_epoch = state.epoch - 1
  return (msg, sending_epoch, output_key)

def Ct1Received.Receive(state, msg):
  output_key = None
  receiving_epoch = state.epoch - 1

  if msg.epoch == state.epoch and msg.type == Ct2:
      # Initialize ct2 decoder
      ct2_decoder = Decoder.new(KEM.CT2_SIZE + MAC_SIZE)
      ct2_decoder.add_chunk(msg.data)

      # Update state
      # Transition (4)
      state = EkSentCt1Received(
        state.epoch,
        state.auth,
        state.dk,
        state.ct1,
        ct2_decoder)

  return (receiving_epoch, output_key)
```

EkSentCt1Received

Represents an agent that has received *ct1*, sent *ek*, and is receiving chunks of *ct2*. Additional state includes:

- *dk*: a KEM decapsulation key
- *ct1*: The compressed public key part of a KEM ciphertext
- *ct2_decoder*

In the **EkSentCt1Received** state an agent doesn't send any data to the other party and it receives chunks of *ct2*. Once *ct2* is received, it verifies the MAC, decapsulates the secret, emits the key, and transitions to the **NoHeaderReceived** state to wait for the other party to begin sending an encapsulation key for the next epoch:

```
def EkSentCt1Received.Send(state):
  # No data to send
  msg = {epoch: state.epoch, type: None}

  # Return values
  output_key = None
  sending_epoch = state.epoch - 1
  return (msg, sending_epoch, output_key)

def EkSentCt1Received.Receive(state, msg):
  output_key = None
  receiving_epoch = state.epoch - 1

  if msg.epoch == state.epoch and msg.type == Ct2:
      # Add chunk to decoder
      state.ct2_decoder.add_chunk(msg.data)

      # Check if ct2 is complete
      if state.ct2_decoder.has_message():
          ct2_with_mac = state.ct2_decoder.message()
          ct2 = ct2_with_mac[:KEM.CT2_SIZE]
          mac = ct2_with_mac[KEM.CT2_SIZE:]

          # Decapsulate shared secret
          ss = KEM.Decaps(state.dk, state.ct1, ct2)
          ss = KDF_OK(ss, state.epoch)

          # Update authenticator and verify MAC
          state.auth.Update(state.epoch, ss)
          state.auth.VfyCt(state.epoch, state.ct1 || ct2, mac)

          # Prepare for next epoch
```

```
            header_decoder = Decoder.new(KEM.HEADER_SIZE + MAC_SIZE)

            # Update state and return key
            # Transition (5)
            state = NoHeaderReceived(
              state.epoch + 1,
              state.auth,
              header_decoder)
            output_key = (state.epoch - 1, ss)

  return (receiving_epoch, output_key)
```

The following describes the state of an agent when they are transmitting a ciphertext in response to an encapsulation key.

NoHeaderReceived

Represents an agent that is receiving a header. Additional state includes:

- *header_decoder*

In the **NoHeaderReceived** state an agent receives chunks of the header. Once the header has been completely received, it transitions to the **HeaderReceived** state, but does not sample the ciphertext yet:

```
def NoHeaderReceived.Send(state):
  # No data to send
  msg = {epoch: state.epoch, type: None}

  # Return values
  output_key = None
  sending_epoch = state.epoch - 1
  return (msg, sending_epoch, output_key)

def NoHeaderReceived.Receive(state, msg):
  output_key = None
  receiving_epoch = state.epoch - 1

  if msg.epoch == state.epoch and msg.type == Hdr:
      # Add chunk to decoder
      state.header_decoder.add_chunk(msg.data)

      # Check if header is complete
      if state.header_decoder.has_message():
          header_with_mac = state.header_decoder.message()
          header = header_with_mac[:64]
          mac = header_with_mac[64:]
          ek_seed = header[:32]
```

```
            hek = header[32:]

            # Verify header MAC
            state.auth.VfyHdr(state.epoch, header, mac)

            # Prepare ek_vector decoder
            ek_decoder = Decoder.new(KEM.EK_SIZE)

            # Update state
            # Transition (6)
            state = HeaderReceived(
              state.epoch,
              state.auth,
              ek_seed,
              hek,
              ek_decoder)

  return (receiving_epoch, output_key)
```

HeaderReceived

Represents an agent that has received a header and is prepared to sample a new *ct1* on the next send. Additional state includes:

- *ek_seed*: seed of a KEM encapsulation key
- *hek*: SHA3 hash of *ek_seed* // *ek_vector*
- *ek_decoder*

In the **HeaderReceived** state an agent is ready to sample a ciphertext when asked to send. When it does this, it computes the encapsulated shared secret for this epoch and returns it to the caller. While it has an *ek_decoder* prepared, it will not receive any *ek_vector* chunks until after it has sent a *ct1* message - and then it will have transitioned out of this state. So the *Receive* function is a no-op:

```
def HeaderReceived.Send(state):
  # Generate shared secret and ct1
  (encaps_secret, ct1, ss) = KEM.Encaps1(state.ek_seed, state.hek)
  ss = KDF_OK(ss, state.epoch)

  # Update authenticator
  state.auth.Update(state.epoch, ss)

  # Encode ct1 for transmission
  ct1_encoder = Encode(ct1)
  chunk = ct1_encoder.next_chunk()
  msg = {epoch: state.epoch, type: Ct1, data: chunk}
```

```
  # Update state
  # Transition (7)
  state = Ct1Sampled(
    state.epoch,
    state.auth,
    state.ek_seed,
    state.hek,
    encaps_secret,
    ct1,
    ct1_encoder,
    state.ek_decoder)

  # Return values
  output_key = (state.epoch, ss)
  sending_epoch = state.epoch - 1
  return (msg, sending_epoch, output_key)

def HeaderReceived.Receive(state, msg):
  # No action taken
  output_key = None
  receiving_epoch = state.epoch - 1
  return (receiving_epoch, output_key)
```

Ct1Sampled

Represents an agent that has received a header, has sampled *ct1*, and is sending it in chunks. Additional state includes:

- *ek_seed*: seed of a KEM encapsulation key
- *hek*: SHA3 hash of *ek_seed* // *ek_vector*
- *encaps_secret*: the secret material used to encapsulate a KEM ciphertext
- *ct1*: The compressed public key part of a KEM ciphertext
- *ct1_encoder*
- *ek_decoder*

The **Ct1Sampled** state has the most complex transition possibilities. In this state an agent is receiving chunks of *ek_vector* and sending chunks of *ct1*. If it receives all of *ek_vector* before receiving an acknowledgment that *ct1* was received, it will transition to **EkReceivedCt1Sampled**. On the other hand, if it receives an acknowledgment that *ct1* was received before *ek_vector* has been completely received, it will transition to **Ct1Acknowledged**. If this agent both receives an acknowledgment for *Ct1* and receives the last chunk of *ek_vector* in a single receive call, it will compute *ct1* and transition to **Ct2Sampled**:

```
def Ct1Sampled.Send(state):
  # Generate next ct1 chunk
```

```
  chunk = state.ct1_encoder.next_chunk()
  msg = {epoch: state.epoch, type: Ct1, data: chunk}

  # Return values
  output_key = None
  sending_epoch = state.epoch - 1
  return (msg, sending_epoch, output_key)

def Ct1Sampled.Receive(state, msg):
  output_key = None
  receiving_epoch = state.epoch - 1

  if msg.epoch == state.epoch and msg.type == Ek:
      # Add ek_vector chunk
      state.ek_decoder.add_chunk(msg.data)

      # Check if ek_vector is complete
      if state.ek_decoder.has_message():
          ek_vector = state.ek_decoder.message()

          # Verify ek_vector integrity
          if SHA3-256(state.ek_seed || ek_vector) != state.hek:
              raise Error("EK integrity check failed")

          # Update state
          # Transition (10)
          state = EkReceivedCt1Sampled(
            state.epoch,
            state.auth,
            state.encaps_secret,
            state.ct1,
            state.ek_seed,
            ek_vector,
            state.ct1_encoder)

  elif msg.epoch == state.epoch and msg.type == EkCt1Ack:
      # Add ek_vector chunk (with acknowledgment)
      state.ek_decoder.add_chunk(msg.data)

      # Check if ek_vector is complete
      if state.ek_decoder.has_message():
          ek_vector = state.ek_decoder.message()

          # Verify ek_vector integrity
          if SHA3-256(state.ek_seed || ek_vector) != state.hek:
              raise Error("EK integrity check failed")
```

```
        # Complete encapsulation
        ct2 = KEM.Encaps2(
          state.encaps_secret, state.ek_seed, ek_vector)
        mac = state.auth.MacCt(state.epoch, state.ct1 || ct2)
        ct2_encoder = Encode(ct2 || mac)

        # Update state
        # Transition (9)
        state = Ct2Sampled(state.epoch, state.auth, ct2_encoder)
    else:
        # Update state
        # Transition (8)
        state = Ct1Acknowledged(
          state.epoch,
          state.auth,
          state.encaps_secret,
          state.ek_seed,
          state.hek,
          state.ct1,
          state.ek_decoder)

  return (receiving_epoch, output_key)
```

EkReceivedCt1Sampled

Represents an agent that has received an encapsulation key and is still sending *ct1* in chunks. Additional state includes:

- *encaps_secret*: the secret material used to encapsulate a KEM ciphertext
- *ct1*: The compressed public key part of a KEM ciphertext
- *ek_seed*
- *ek_vector*
- *ct1_encoder*

In the **EkReceivedCt1Sampled** state an agent sends chunks of *ct1* and awaits an acknowledgment that it has been received. When that acknowledgment comes, it computes *ct2* and transitions to the **Ct2Sampled** state:

```
def EkReceivedCt1Sampled.Send(state):
  # Generate next ct1 chunk
  chunk = state.ct1_encoder.next_chunk()
  msg = {epoch: state.epoch, type: Ct1, data: chunk}

  # Return values
  output_key = None
  sending_epoch = state.epoch - 1
```

```
  return (msg, sending_epoch, output_key)

def EkReceivedCt1Sampled.Receive(state, msg):
  output_key = None
  receiving_epoch = state.epoch - 1

  if msg.epoch == state.epoch and msg.type == EkCt1Ack:
      # Complete encapsulation
      ct2 = KEM.Encaps2(
        state.encaps_secret, state.ek_seed, state.ek_vector)
      mac = state.auth.MacCt(state.epoch, state.ct1 || ct2)
      ct2_encoder = Encode(ct2 || mac)

      # Update state
      # Transition (12)
      state = Ct2Sampled(state.epoch, state.auth, ct2_encoder)

  return (receiving_epoch, output_key)
```

Ct1Acknowledged

Represents an agent that has completed sending *ct1* but is still receiving chunks of *ek_vector*. Additional state includes:

- *ek_seed*: seed of a KEM encapsulation key
- *hek*: SHA3 hash of *ek_seed* // *ek_vector*
- *encaps_secret*: the secret material used to encapsulate a KEM ciphertext
- *ct1*: The compressed public key part of a KEM ciphertext
- *ek_decoder*

In the **Ct1Acknowledged** state an agent receives chunks of an incoming *ek_vector*. Once this has been completely received, it can compute *ct2* and transition to the **Ct2Sampled** state:

```
def Ct1Acknowledged.Send(state):
  # No data to send
  msg = {epoch: state.epoch, type: None}

  # Return values
  output_key = None
  sending_epoch = state.epoch - 1
  return (msg, sending_epoch, output_key)

def Ct1Acknowledged.Receive(state, msg):
  output_key = None
  receiving_epoch = state.epoch - 1
```

```
  if msg.epoch == state.epoch and msg.type == EkCt1Ack:
      # Add ek_vector chunk
      state.ek_decoder.add_chunk(msg.data)

      # Check if ek_vector is complete
      if state.ek_decoder.has_message():
          ek_vector = state.ek_decoder.message()

          # Verify ek_vector integrity
          if SHA3-256(state.ek_seed || ek_vector) != state.hek:
              raise Error("EK integrity check failed")

          # Complete encapsulation
          ct2 = KEM.Encaps2(
            state.encaps_secret, state.ek_seed, ek_vector)
          mac = state.auth.MacCt(state.epoch, state.ct1 || ct2)
          ct2_encoder = Encode(ct2 || mac)

          # Update state
          # Transition (11)
          state = Ct2Sampled(state.epoch, state.auth, ct2_encoder)

  return (receiving_epoch, output_key)
```

Ct2Sampled

Represents an agent that has completed sending *ct1*, received *ek_vector*, and is sending *ct2*. Additional state includes:

- *ct2_encoder*

In the **Ct2Sampled** state an agent sends chunks of *ct2* and waits for a message from the next epoch. Once a message from the next epoch is received, it transitions to the **KeysUnsampled** state and prepares to start sending a new encapsulation key:

```
def Ct2Sampled.Send(state):
  # Generate next ct2 chunk
  chunk = state.ct2_encoder.next_chunk()
  msg = {epoch: state.epoch, type: Ct2, data: chunk}

  # Return values
  output_key = None
  sending_epoch = state.epoch - 1
  return (msg, sending_epoch, output_key)

def Ct2Sampled.Receive(state, msg):
```

```
output_key = None

if msg.epoch == state.epoch + 1:
    # Next epoch has begun
    # Transition (13)
    state = KeysUnsampled(state.epoch + 1, state.auth)

receiving_epoch = state.epoch - 1
return (receiving_epoch, output_key)
```

2.6 Initialization

We initialize Alice and Bob's protocol state using a preshared secret that may come from a handshake protocol such as PQXDH [7]. Alice is initialized to begin sending an encapsulation key header, while Bob is initialized to expect to receive that header:

```
def InitAlice(shared_secret):
    epoch = 1
    auth = Authenticator.Init(epoch, shared_secret)
    return KeysUnsampled(epoch, auth)

def InitBob(shared_secret):
    epoch = 1
    auth = Authenticator.Init(epoch, shared_secret)
    header_decoder = Decoder.new(KEM.HEADER_SIZE + MAC_SIZE)
    return NoHeaderReceived(epoch, auth, header_decoder)
```

With this initialization, Alice and Bob will always be able to make forward progress as long as fresh messages are delivered. The graph of possible state transitions can be seen in the figure below.

3. Security Considerations

3.1 The Vulnerable Message Set

This protocol is designed to provide robust continuous key agreement in the presence of bandwidth limits. Since many messages must be passed in order to reach key agreement, the important measure of security provided by this protocol is "in the event of a compromise, how many messages are passed before healing?". We call this the *vulnerable message set*.

The size of the vulnerable message set is not an intrinsic property of the protocol. For the ML-KEM Braid, given a chunk size and choice of KEM one can compute the *minimum* possible size of the vulnerable message set, but the *maximum* size of this set is unbounded: if Bob never replies to Alice but Alice continues

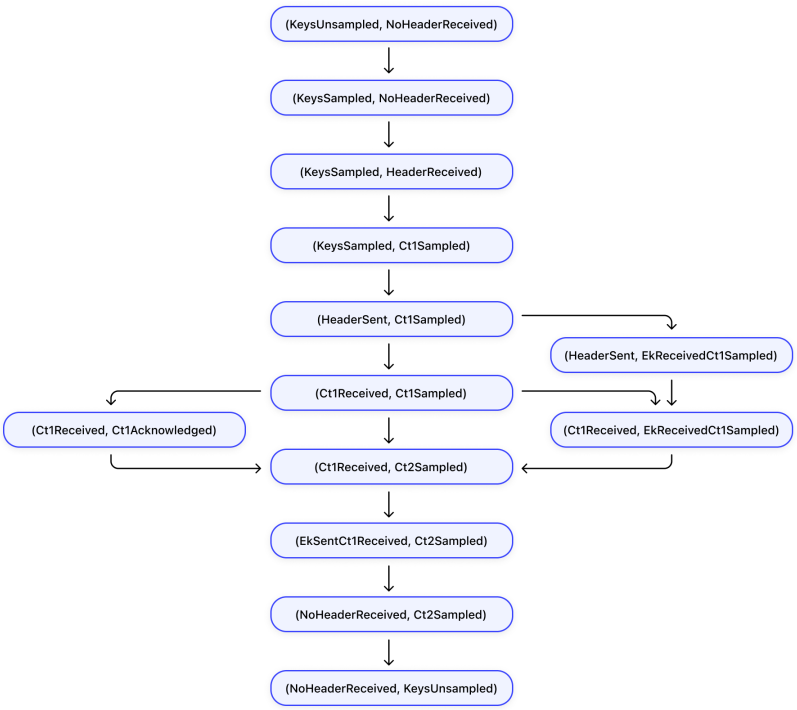

Figure 2: The graph of all possible state transitions for Alice and Bob when Alice begins in the KeysUnsampled state and Bob begins in the NoHeaderReceived state. In each tuple, Alice's state is on the left and Bob's state is on the right. At the end of this process, Alice and Bob will have switched states and will have advanced one epoch.

sending messages, their session will never heal and all of Alice's messages will be vulnerable.

In fact, as shown in [3], the size of the vulnerable message set for a SCKA protocol depends on the message sending behavior of the protocol participants. Two parties that are online and in rapid conversation, such as two parties chatting using their primary devices in a Signal chat, will typically have a smaller vulnerable message set than will be found in a conversation between two parties on desktop devices that are often offline.

The ML-KEM Braid protocol was selected to provide small vulnerable message sets in a wide range of realistic secure messaging scenarios, but for applications with highly specific message sending behaviors protocol designers should consider whether a different SCKA protocol may provide better security.

3.2 Alternate KEMs

This protocol uses ML-KEM as specified but we note that the IND-CCA security provided by ML-KEM's Fujisaki-Okamoto transform is not required to prove the security of this or related SCKA protocols. In [8] an IND-CPA "Ratcheting KEM" was designed with this fact in mind, and it allows a large part of a ciphertext to be reused as a public key for a future round, reducing bandwidth costs. Even without the ratcheting KEM optimization, this protocol could be made more efficient by having the parties use their shared session state to determine the encapsulation key seed, and then using the internal IND-CPA Public Key Encryption (PKE) functionality of ML-KEM. This would not only reduce message sizes by a small amount, it would allow us to skip sending the "header" and reduce the number of round trips required to emit a key. This could have a particularly large benefit in situations where communication is imbalanced, for example, when one party's device is offline for long periods of time.

Alternate KEMs, or more generally IND-CPA PKE schemes designed for efficient ratcheting, may not have the same binding properties as ML-KEM. As seen in [9], [10], these binding properties can have an impact on the security of higher level protocols. Developers using a variant of this protocol with alternate KEM should consider the security implications of these binding properties in their higher level protocol.

3.3 Optional internal authentication

The authenticator objects and the MACs that are added to header and ciphertext messages provide standalone authenticity guarantees for ML-KEM Braid messages and outputs at a cost of transmitting 64 bytes per epoch.

If this protocol is integrated into a higher level protocol, such as the Double Ratchet [2], that provides authentication, then protocol messages can derive

authenticity from that and the internal authentication of the ML-KEM Braid protocol could be removed.

3.4 Bandwidth limits, message sizes, and speed of PCS

While the size of the vulnerable message set depends on the actual message sending behavior of the protocol participants, it is easy to see that if we send larger chunks, then key agreement - and eventual healing - will happen faster.

With a chunk size of 32, using ML-KEM 768 we require 3 messages to send a header, 30 messages to send *ct1*, 36 messages to send *ek_vector*, and 5 messages to send *ct2* with its MAC. If we doubled the chunk size to 64, these numbers will be cut in half (with upward rounding) and, under ideal conditions, healing would occur almost twice as fast. We note, though, that when parties go offline for periods of time, as is common with Signal's linked devices, the benefits of larger chunks become less pronounced, as is seen in [3]. A doubling of the bandwidth limit will significantly improve the speed of PCS healing in many realistic settings but it will not lead to a doubling of healing speed. When considering the tradeoff between bandwidth usage and PCS speed, both client requirements and client message sending behavior should be considered.

In the presence of strict bandwidth limits, note that if a compact binary format is used to encode protocol messages and this format saves several bytes over a general purpose format, then those bytes can be used to send larger chunks and speed PCS.

3.5 Encoder domain size

While in principle we think of encoders as producing an unbounded stream of codewords, in practice the encoder we recommend produces a finite number of distinct codewords before repeating. This means that an attacker with network control does not need to perform a complete denial of service attack on two parties to prevent the protocol from advancing: eventually codewords must be repeated and they can let any messages that repeat codewords through.

Internally, this is due to the fact that the encoding is implemented using polynomial interpolation over a finite field, and the number of distinct codewords is equal to the size of the underlying field.

We recommend using $GF(2^{16})$ as the underlying field. When using ML-KEM 768 [1] as the underlying KEM and 32-byte codewords, the largest message sent by the protocol is 36 codewords long. Thus to prevent ratcheting, an attacker must prevent 2^{16} - 35 out of every 2^{16} messages from being delivered. While this is not a complete denial of service, it requires blocking over 99.9% of messages and will likely cause protocol users to consider the underlying service to be unavailable.

Using $GF(2^8)$ would allow faster encoding and decoding, but now would allow an attacker to prevent the protocol from advancing by blocking 221 out of 256

messages, or 86% of messages. This will likely lead to a poor user experience, but may still render a higher level protocol usable. If encoding and decoding speed are critical for an application, we recommend considering the use of *fountain codes* as described in the next section.

3.6 Alternate encoders

We recommend using a systematic erasure code based encoding scheme to split large messages into fixed size chunks. Systematic encodings make the common case where no messages are dropped very efficient, since decoding is simply concatenation. Erasure codes give us a guarantee that, if a message's plaintext fits into N codewords, then when the recipient receives any N codewords they will be able to decode the message.

If the computational costs of erasure code decoding are too high, a *fountain code*, such as RaptorQ [11], can be used. With a fountain code the recipient loses the *guarantee* that if a message's plaintext fits into N codewords, then when the recipient receives any N codewords they will be able to decode the message. Nevertheless, the recipient will still be able to decode the message from N codewords in most situations. For RaptorQ, for example, decoders must succeed in decoding from N codewords at least 99 out of 100 times, and succeed decoding from $N+k$ codewords with probability at least $1 - 10^{-k-1}$.

Since four messages must be decoded in each epoch, we can expect an implementation using RaptorQ to have to send more messages in an epoch than an erasure code based implementation about 4% of the time.

This may have a negative impact on the security of any higher level protocol. For example, in a Double Ratchet protocol [2] using this would directly lead to an increase in the size of the Vulnerable Message Set - the set of messages exposed to an attacker in a device compromise - about 4% of the time. (See [3] for more details on the Vulnerable Message Set).

3.7 Formal verification and security proofs

The ML-KEM Braid protocol has been modeled using ProVerif [12] and has been proven, in the Dolev-Yao model, to provide the correctness, Forward Secrecy, and Post-Compromise Security required of an SCKA protocol as defined in [3]. Furthermore these models also prove mutual authentication for the ML-KEM Braid protocol.

The ML-KEM Braid protocol is closely related to the protocol Opp-UniKEM-CKA introduced and proven to be a secure SCKA in [3].

The implementation and ProVerif models are available at [13].

3.8 Representation of epochs

If an implementation uses a fixed width integer to represent the epoch, then eventually the epoch counter will repeat and the protocol loses the property that every epoch produces a unique key. For example, an application that uses 8-bit integers for epochs to conserve space in protocol messages will begin repeating epochs after 256 keys have been emitted - something that is likely to happen in many conversations. Using a 64-bit integer to represent the epoch will prevent this wraparound from ever happening in a human conversation, but for other applications of the ML-KEM Braid this wraparound should be considered.

4. IPR

5. Acknowledgements

The ML-KEM Braid protocol was designed by Graeme Connell and Rolfe Schmidt.

The notion of Sparse Continuous Key Agreement and the general opportunistic sending strategy underlying the ML-KEM Braid were introduced in [3]. The co-authors of this paper, Benedikt Auerbach, Yevgeniy Dodis, Daniel Jost, and Shuichi Katsumata contributed protocol design, analysis, and editorial feedback.

Karthik Bhargavan and Franziskus Kiefer were involved throughout the implementation process and contributed to the detailed design, modeling and analysis of the ML-KEM Braid using ProVerif, and provided editorial feedback on this documentation.

Thanks to Rune Fiedler, Charlie Jacomme and Nadim Kobeissi for valuable editorial feedback.

This work builds on the firm foundation the cryptography research community has created for us, and we deeply appreciate their continued efforts to improve our understanding of secure communication.

[1] "Module-lattice-based key-encapsulation mechanism standard." https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.ipd.pdf

[2] T. Perrin and M. Marlinspike, "The Double Ratchet Algorithm," 2016. https://signal.org/docs/specifications/doubleratchet/

[3] B. Auerbach, Y. Dodis, D. Jost, S. Katsumata, and R. Schmidt, "How to compare two-party secure messaging protocols: A quest for a more efficient and secure post-quantum protocol," 2025.

[4] E. Fujisaki and T. Okamoto, "Secure integration of asymmetric and symmetric encryption schemes," J. Cryptol., vol. 26, no. 1, 2013. https://doi.org/10.1007/s00145-011-9114-1

[5] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)." Internet Engineering Task Force; RFC 5869 (Informational); IETF, May-2010. http://www.ietf.org/rfc/rfc5869.txt

[6] "Protocol buffers." https://protobuf.dev/

[7] E. Kret and R. Schmidt, "The PQXDH key agreement protocol," 2023. https://signal.org/docs/specifications/pqxdh/

[8] Y. Dodis, D. Jost, S. Katsumata, T. Prest, and R. Schmidt, "Triple ratchet: A bandwidth efficient hybrid-secure signal protocol." Cryptology ePrint Archive, Paper 2025/078, 2025. https://eprint.iacr.org/2025/078

[9] K. Bhargavan, C. Jacomme, and F. Kiefer, "PQXDH formal analysis git repository." https://github.com/Inria-Prosecco/pqxdh-analysis

[10] C. J. F. Cremers, A. Dax, and N. Medinger, "Keeping up with the KEMs: Stronger security notions for KEMs and automated analysis of KEM-based protocols," in Conference on computer and communications security, 2024. https://api.semanticscholar.org/CorpusID:268289969

[11] M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, and L. Minder, "RaptorQ Forward Error Correction Scheme for Object Delivery." Internet Engineering Task Force; RFC 6330 (Proposed Standard); IETF, Aug-2011. http://www.ietf.org/rfc/rfc6330.txt

[12] "ProVerif." https://bblanche.gitlabpages.inria.fr/proverif/

[13] S. Messenger, "Sparse Post-Quantum Ratchet," 2025. https://github.com/signalapp/sparsepostquantumratchet