The Double Ratchet Algorithm

Trevor Perrin (editor) Moxie Marlinspike
Rolfe Schmidt (revision 3+)

Revision 4 , 2025-11-04

Contents

1. Introduction **3**

2. Overview **3**
2.1. KDF chains . . . 3
2.2. Symmetric-key ratchet . . . 5
2.3. Diffie-Hellman ratchet . . . 6
2.4. Double Ratchet . . . 13
2.6. Out-of-order messages . . . 17

3. Double Ratchet **18**
3.1. External functions . . . 18
3.2. State variables . . . 19
3.3. Initialization . . . 19
3.4. Encrypting messages . . . 20
3.5. Decrypting messages . . . 20

4. Double Ratchet with header encryption **23**
4.1. Overview . . . 23
4.2. External functions . . . 27
4.3. State variables . . . 27
4.4. Initialization . . . 27
4.5. Encrypting messages . . . 28
4.6. Decrypting messages . . . 29

5. The Sparse Post-Quantum Ratchet **30**
5.1 Sparse Continuous Key Agreement . . . 31
5.2 External functions . . . 32
5.3 State variables . . . 32
5.4 Initialization . . . 32
5.5 Encrypting Messages . . . 33
5.6 Decrypting Messages . . . 34

5.7 Clearing past epoch state 35

6. The Triple Ratchet: combining Secure Messaging protocols for hybrid security 37
6.1 Overview 37
6.2 State variables 37
6.3 External functions 37
6.4 Initialization 37
6.5 Encrypting Messages 38
6.6 Decrypting Messages 38

7. Implementation considerations 38
7.1. Integration with PQXDH 38
7.2. Recommended cryptographic algorithms 40

8. Security considerations 41
8.1. Secure deletion 41
8.2. Recovery from compromise 42
8.3. Cryptanalysis and ratchet public keys 42
8.4. Deletion of skipped message keys 42
8.5. Deferring new ratchet key generation 43
8.6. Truncating authentication tags 43
8.7. Implementation fingerprinting 43
8.8 Choice of SCKA protocol 43
8.9 Effect of dropped messages on PCS 44
8.10 Deletion of old KDF Chain state 44
8.11 Harvest Now, Decrypt Later Attacks 44

9. IPR 44

10. Acknowledgements 44

9. References 45

1. Introduction

The Double Ratchet algorithm is used by two parties to exchange encrypted messages based on a shared secret key. Typically the parties will use some key agreement protocol (such as PQXDH [1]) to agree on the shared secret key. Following this, the parties will use the Double Ratchet to send and receive encrypted messages.

The parties derive new keys for every Double Ratchet message so that earlier keys cannot be calculated from later ones. The parties also send Diffie-Hellman public values attached to their messages. The results of Diffie-Hellman calculations are mixed into the derived keys so that later keys cannot be calculated from earlier ones. These properties gives some protection to earlier or later encrypted messages in case of a compromise of a party's keys.

The Double Ratchet and its header encryption variant are presented below, and their security properties are discussed.

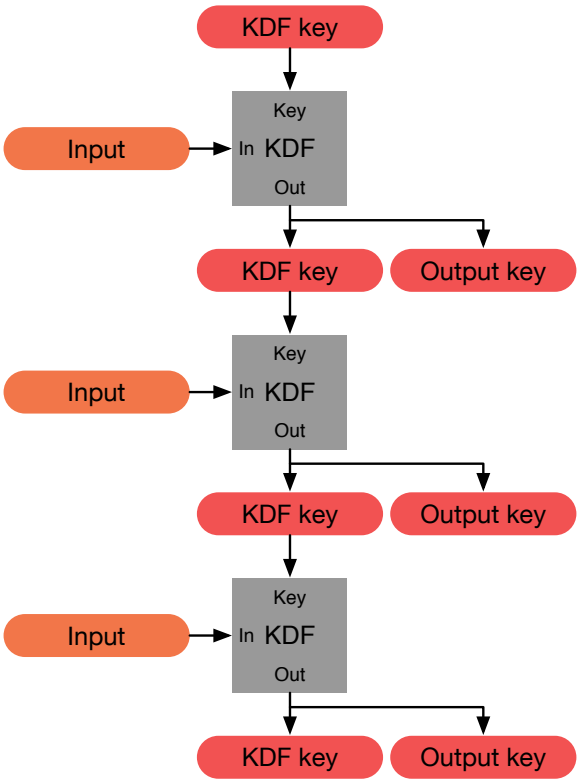
2. Overview

2.1. KDF chains

A **KDF chain** is a core concept in the Double Ratchet algorithm.

We define a **KDF** as a cryptographic function that takes a secret and random **KDF key** and some input data and returns output data. The output data is indistinguishable from random provided the key isn't known (i.e. a KDF satisfies the requirements of a cryptographic "PRF"). If the key is not secret and random, the KDF should still provide a secure cryptographic hash of its key and input data. The HMAC and HKDF constructions, when instantiated with a secure hash algorithm, meet the KDF definition [2], [3].

We use the term **KDF chain** when some of the output from a KDF is used as an **output key** and some is used to replace the KDF key, which can then be used with another input. The below diagram represents a KDF chain processing three inputs and producing three output keys:



A KDF chain has the following properties (using terminology adapted from [4]):

- **Resilience:** The output keys appear random to an adversary without knowledge of the KDF keys. This is true even if the adversary can control the KDF inputs.
- **Forward security:** Output keys from the past appear random to an adversary who learns the KDF key at some point in time.
- **Break-in recovery:** Future output keys appear random to an adversary who learns the KDF key at some point in time, provided that future inputs have added sufficient entropy.

In a **Double Ratchet session** between Alice and Bob each party stores a KDF key for three chains: a **root chain**, a **sending chain**, and a **receiving chain** (Alice's sending chain matches Bob's receiving chain, and vice versa).

As Alice and Bob exchange messages they also exchange new Diffie-Hellman

public keys, and the Diffie-Hellman output secrets become the inputs to the root chain. The output keys from the root chain become new KDF keys for the sending and receiving chains. This is called the **Diffie-Hellman ratchet**.

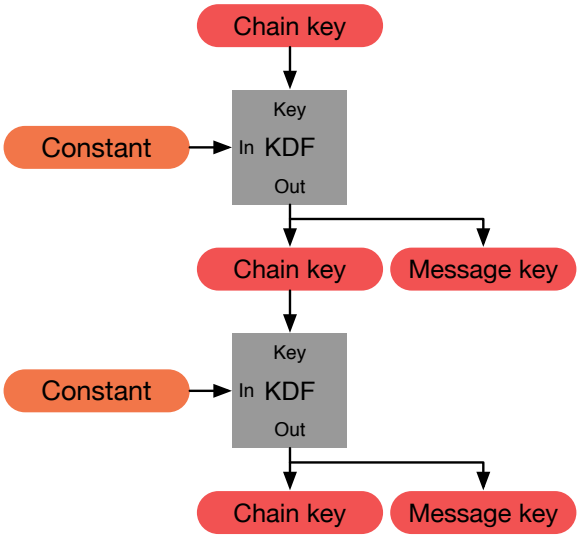
The sending and receiving chains advance as each message is sent and received. Their output keys are used to encrypt and decrypt messages. This is called the **symmetric-key ratchet**

The next sections explain the symmetric-key and Diffie-Hellman ratchets in more detail, then show how they are combined into the Double Ratchet.

2.2. Symmetric-key ratchet

Every message sent or received is encrypted with a unique **message key**. The message keys are output keys from the sending and receiving KDF chains. The KDF keys for these chains will be called **chain keys**.

The KDF inputs for the sending and receiving chains are constant, so these chains don't provide break-in recovery. The sending and receiving chains just ensure that each message is encrypted with a unique key that can be deleted after encryption or decryption. Calculating the next chain key and message key from a given chain key is a single **ratchet step** in the **symmetric-key ratchet**. The below diagram shows two steps:



Because message keys aren't used to derive any other keys, message keys may be stored without affecting the security of other message keys. This is useful for handling lost or out-of-order messages (see Section 2.6).

2.3. Diffie-Hellman ratchet

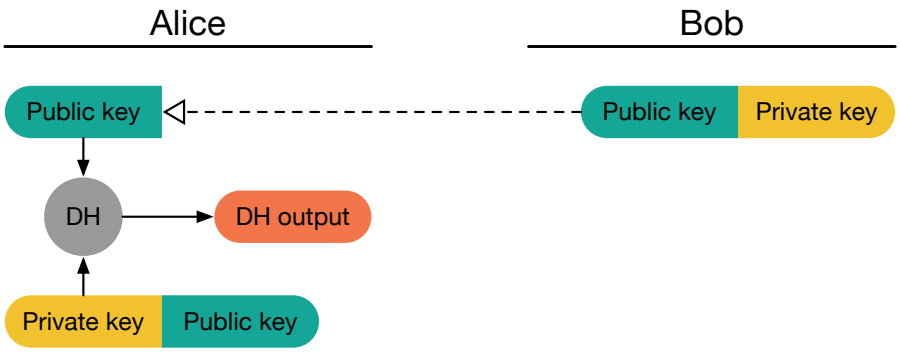
If an attacker steals one party's sending and receiving chain keys, the attacker can compute all future message keys and decrypt all future messages. To prevent this, the Double Ratchet combines the symmetric-key ratchet with a **DH ratchet** which updates chain keys based on Diffie-Hellman outputs.

To implement the DH ratchet, each party generates a DH key pair (a Diffie-Hellman public key and private key) which becomes their current **ratchet key pair**. Every message from either party begins with a header which contains the sender's current ratchet public key. When a new ratchet public key is received from the remote party, a **DH ratchet step** is performed which replaces the local party's current ratchet key pair with a new key pair.

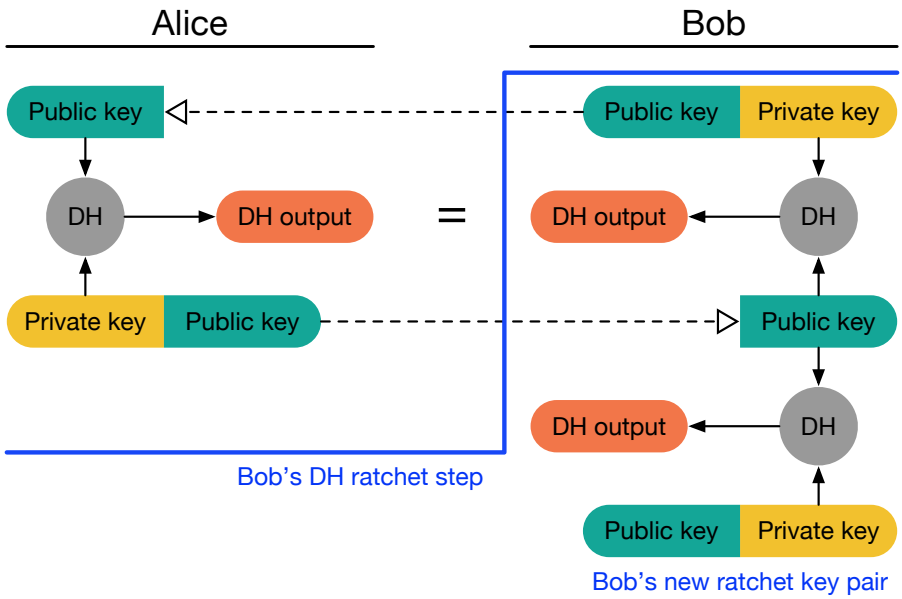
This results in a "ping-pong" behavior as the parties take turns replacing ratchet key pairs. An eavesdropper who briefly compromises one of the parties might learn the value of a current ratchet private key, but that private key will eventually be replaced with an uncompromised one. At that point, the Diffie-Hellman calculation between ratchet key pairs will define a DH output unknown to the attacker.

The following diagrams show how the DH ratchet derives a shared sequence of DH outputs.

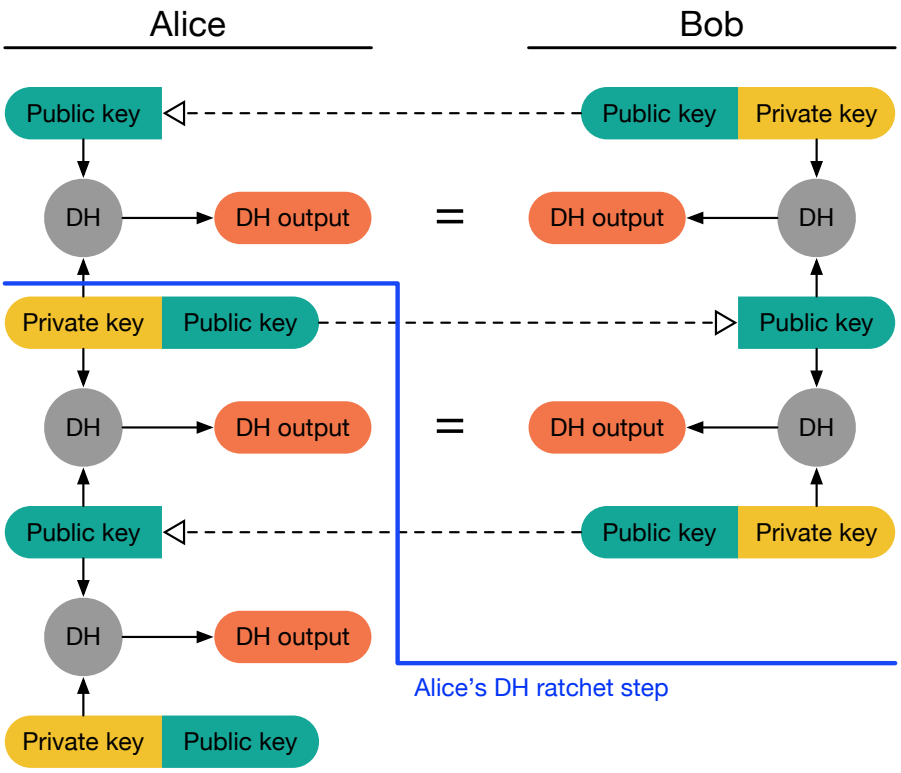
Alice is initialized with Bob's ratchet public key. Alice's ratchet public key isn't yet known to Bob. As part of initialization Alice performs a DH calculation between her ratchet private key and Bob's ratchet public key:



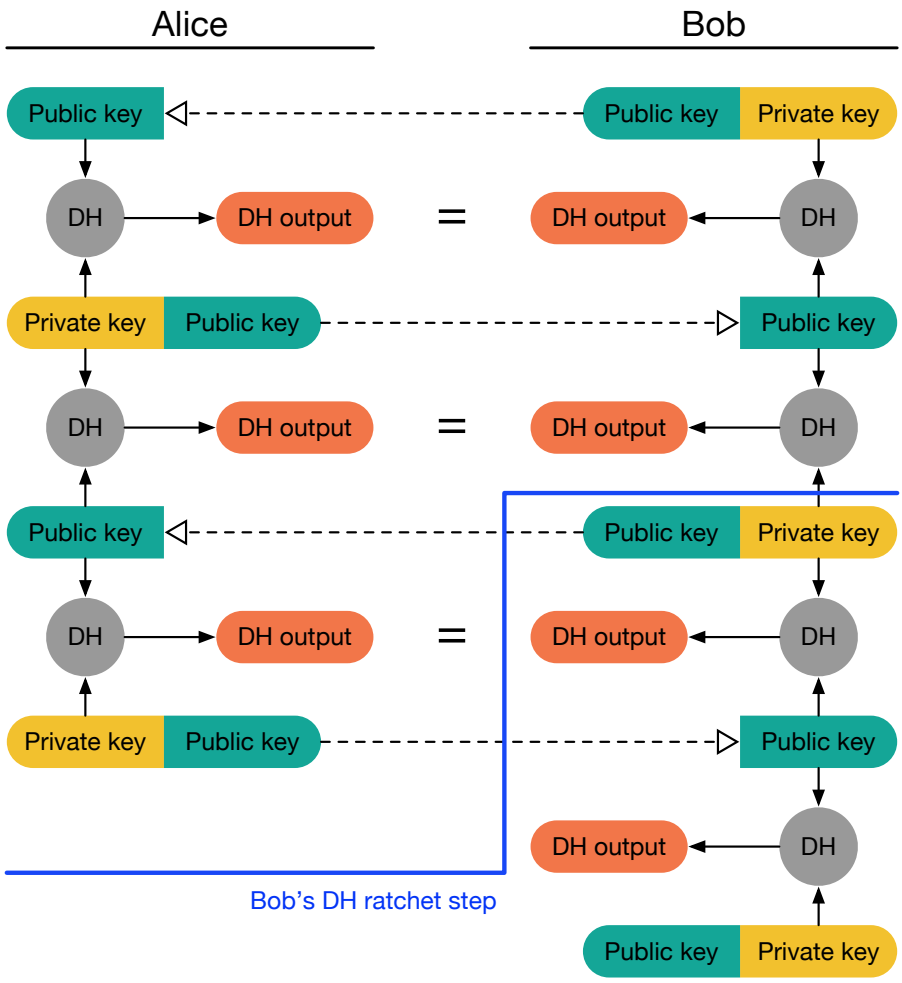
Alice's initial messages advertise her ratchet public key. Once Bob receives one of these messages, Bob performs a DH ratchet step: He calculates the DH output between Alice's ratchet public key and his ratchet private key, which equals Alice's initial DH output. Bob then replaces his ratchet key pair and calculates a new DH output:



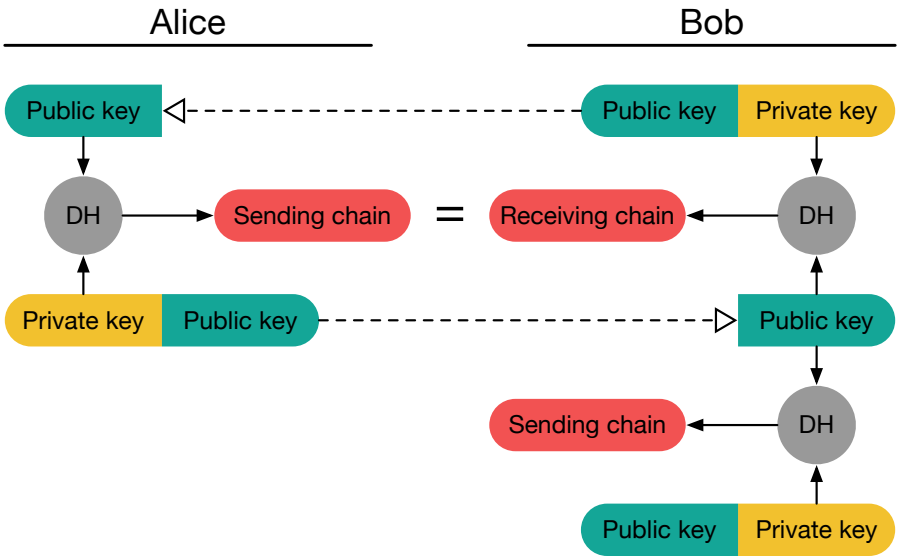
Messages sent by Bob advertise his new public key. Eventually, Alice will receive one of Bob's messages and perform a DH ratchet step, replacing her ratchet key pair and deriving two DH outputs, one that matches Bob's latest and a new one:



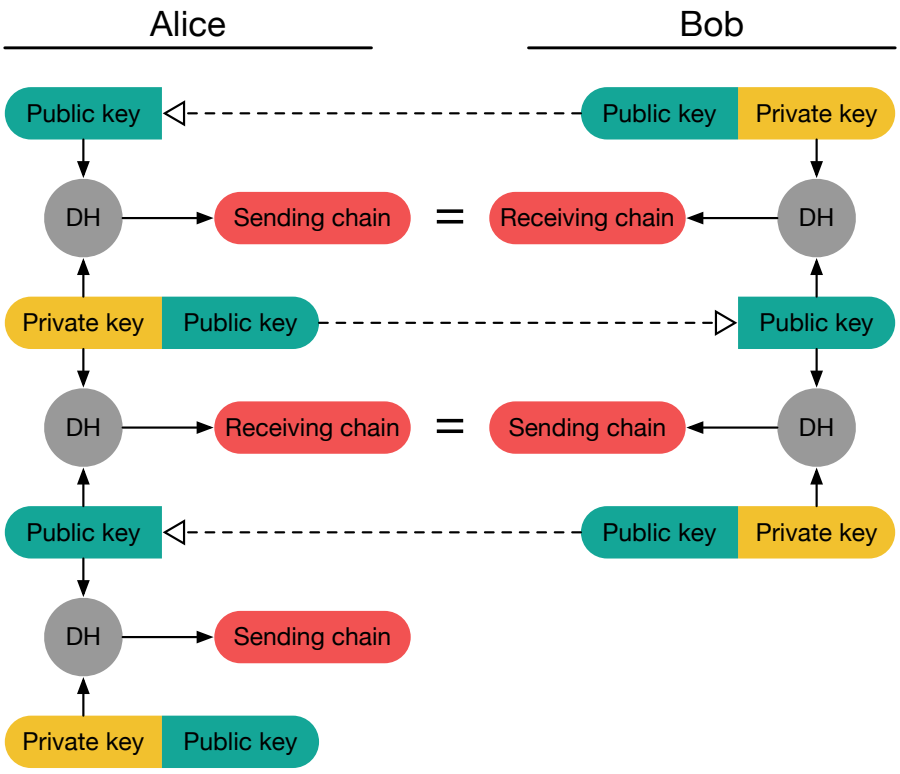
Messages sent by Alice advertise her new public key. Eventually, Bob will receive one of these messages and perform a second DH ratchet step, and so on:



The DH outputs generated during each DH ratchet step are used to derive new sending and receiving chain keys. The below diagram revisits Bob's first ratchet step. Bob uses his first DH output to derive a receiving chain that matches Alice's sending chain. Bob uses the second DH output to derive a new sending chain:

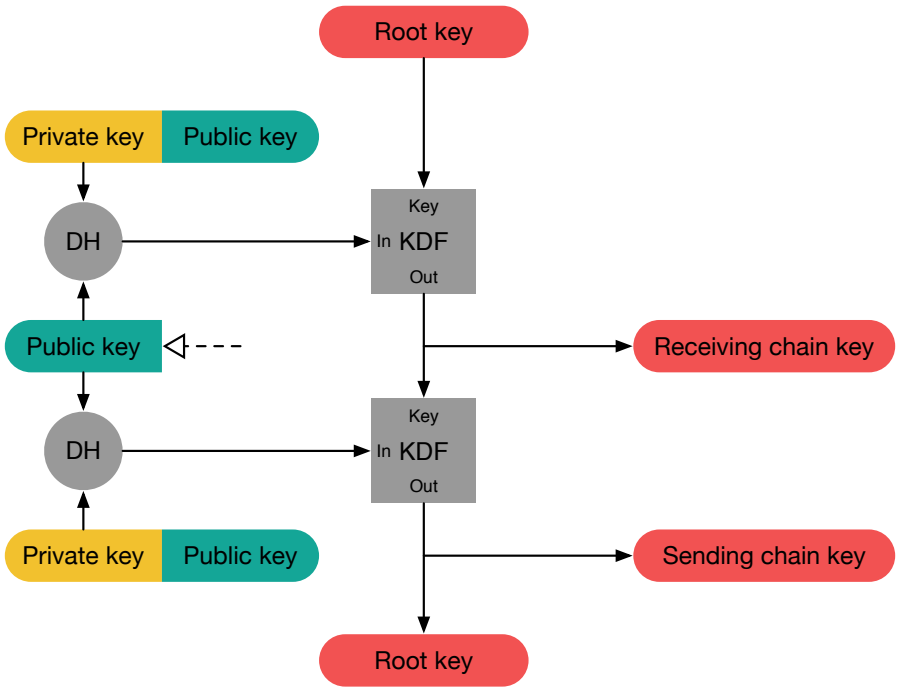


As the parties take turns performing DH ratchet steps, they take turns introducing new sending chains:



However, the above picture is a simplification. Instead of taking the chain keys directly from DH outputs, the DH outputs are used as KDF inputs to a root chain, and the KDF outputs from the root chain are used as sending and receiving chain keys. Using a KDF chain here improves resilience and break-in recovery.

So a full DH ratchet step consists of updating the root KDF chain twice, and using the KDF output keys as new receiving and sending chain keys:

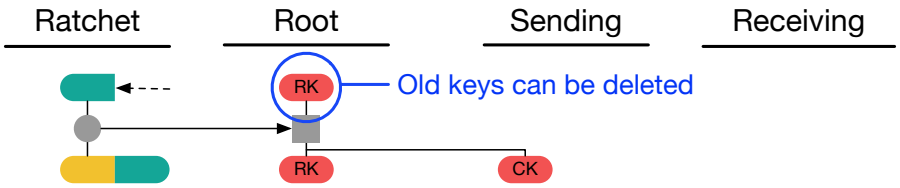


2.4. Double Ratchet

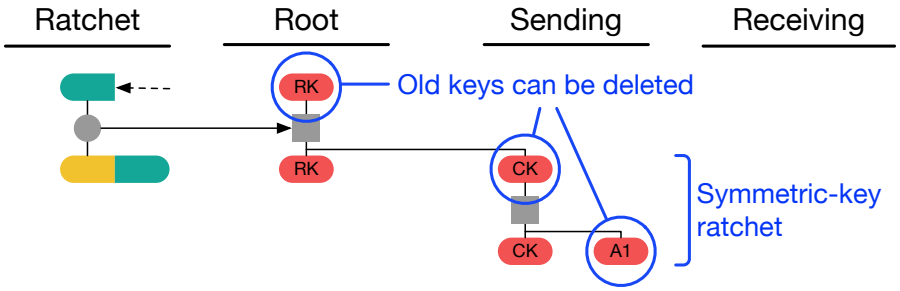
Combining the symmetric-key and DH ratchets gives the Double Ratchet:

- When a message is sent or received, a symmetric-key ratchet step is applied to the sending or receiving chain to derive the message key.
- When a new ratchet public key is received, a DH ratchet step is performed prior to the symmetric-key ratchet to replace the chain keys.

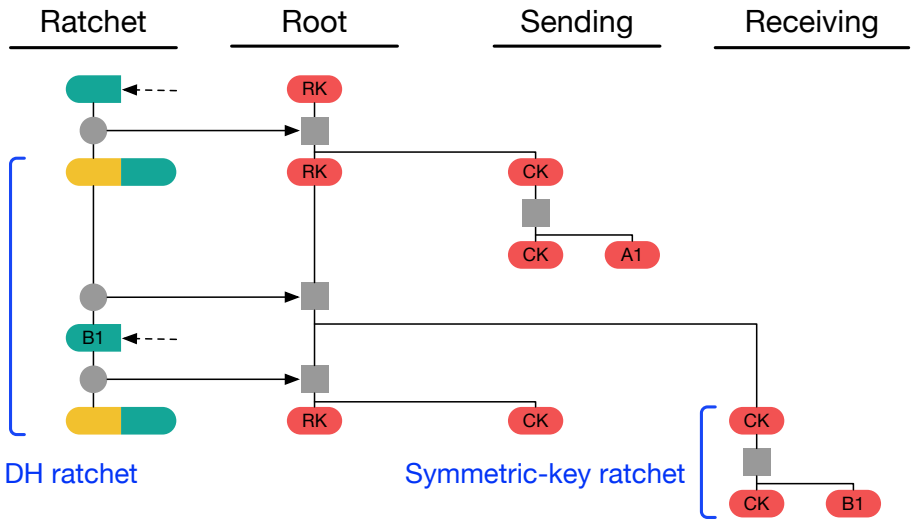
In the below diagram Alice has been initialized with Bob's ratchet public key and a shared secret which is the initial root key. As part of initialization Alice generates a new ratchet key pair, and feeds the DH output to the root KDF to calculate a new root key (RK) and sending chain key (CK):



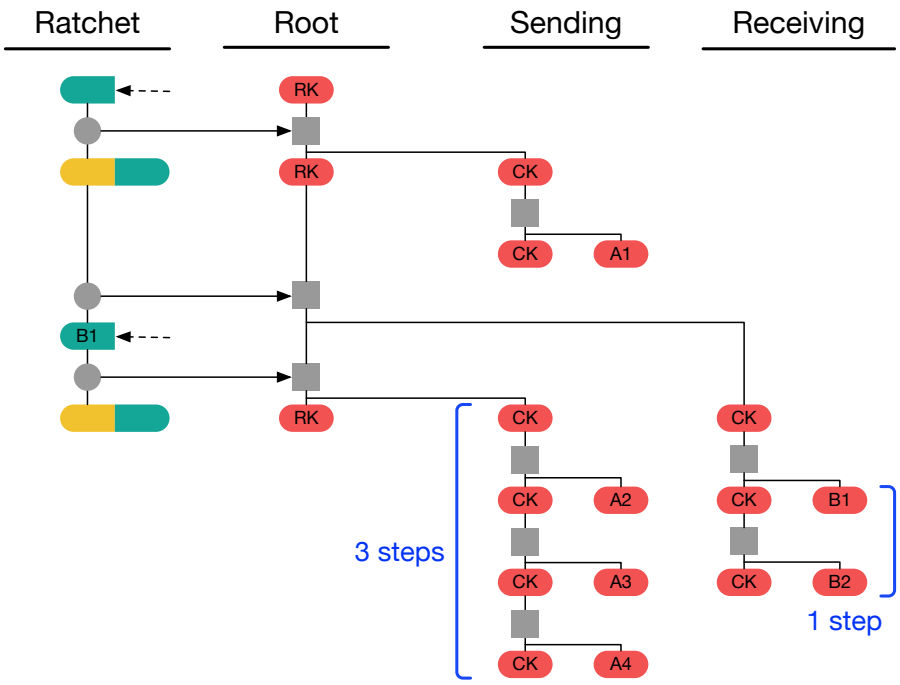
When Alice sends her first message *A1*, she applies a symmetric-key ratchet step to her sending chain key, resulting in a new message key (message keys will be labelled with the message they encrypt or decrypt). The new chain key is stored, but the message key and old chain key can be deleted:



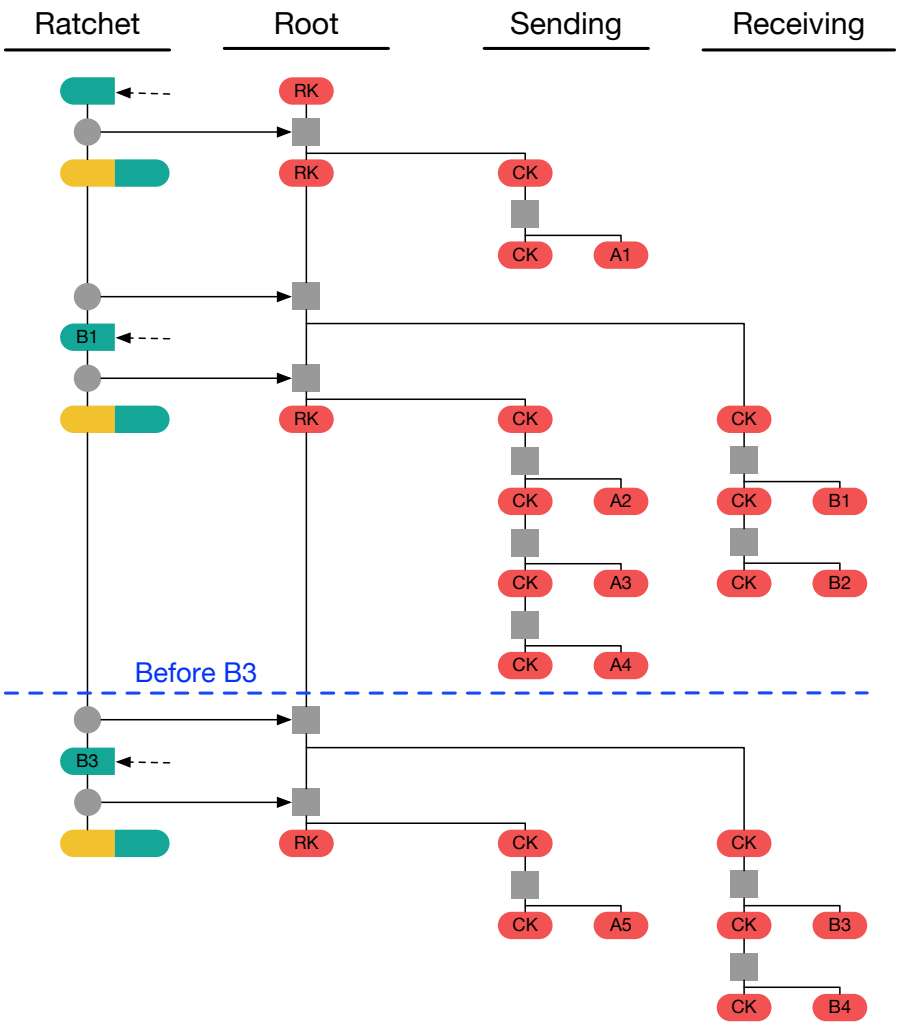
If Alice next receives a response *B1* from Bob, it will contain a new ratchet public key (Bob's public keys are labelled with the message when they were first received). Alice applies a DH ratchet step to derive new receiving and sending chain keys. Then she applies a symmetric-key ratchet step to the receiving chain to get the message key for the received message:



Suppose Alice next sends a message *A2*, receives a message *B2* with Bob's old ratchet public key, then sends messages *A3* and *A4*. Alice's sending chain will ratchet three steps, and her receiving chain will ratchet once:



Suppose Alice then receives messages *B3* and *B4* with Bob's next ratchet key, then sends a message *A5*. Alice's final state will be as follows:



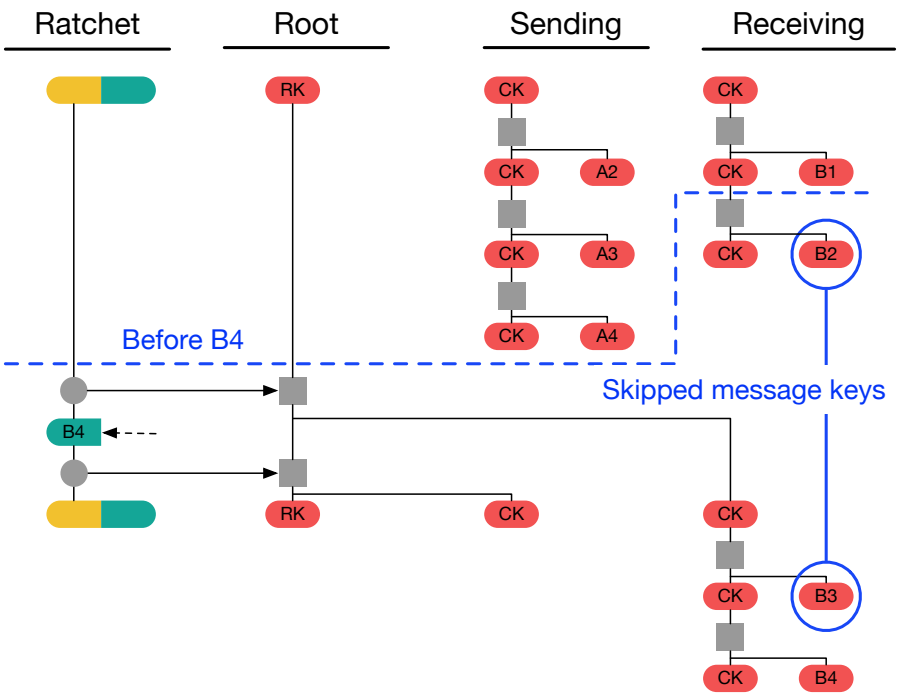
2.6. Out-of-order messages

The Double Ratchet handles lost or out-of-order messages by including in each message header the message's number in the sending chain (N=0,1,2,...) and the length (number of message keys) in the previous sending chain (PN). This enables the recipient to advance to the relevant message key while storing skipped message keys in case the skipped messages arrive later.

On receiving a message, if a DH ratchet step is triggered then the received PN minus the length of the current receiving chain is the number of skipped messages in that receiving chain. The received N is the number of skipped messages in the new receiving chain (i.e. the chain after the DH ratchet).

If a DH ratchet step isn't triggered, then the received N minus the length of the receiving chain is the number of skipped messages in that chain.

For example, consider the message sequence from the previous section when messages *B2* and *B3* are skipped. Message *B4* will trigger Alice's DH ratchet step (instead of *B3*). Message *B4* will have PN=2 and N=1. On receiving *B4* Alice will have a receiving chain of length 1 (*B1*), so Alice will store message keys for *B2* and *B3* so they can be decrypted if they arrive later:



3. Double Ratchet

3.1. External functions

To instantiate the Double Ratchet requires defining the following functions. For recommendations, see Section 7.2.

- ***GENERATE_DH()***: Returns a new Diffie-Hellman key pair.
- ***DH(dh_pair, dh_pub)***: Returns the output from the Diffie-Hellman calculation between the private key from the DH key pair *dh_pair* and the DH public key *dh_pub*. If the DH function rejects invalid public keys, then this function may raise an exception which terminates processing.
- ***KDF_RK(rk, dh_out)***: Returns a pair (32-byte root key, 32-byte chain key) as the output of applying a KDF keyed by a 32-byte root key *rk* to a Diffie-Hellman output *dh_out*.
- ***KDF_CK(ck)***: Returns a pair (32-byte chain key, 32-byte message key) as the output of applying a KDF keyed by a 32-byte chain key *ck* to some constant. If *ck* is *None* this function must fail in a way that terminates processing.
- ***ENCRYPT(mk, plaintext, associated_data)***: Returns an AEAD encryption of *plaintext* with message key *mk* [5]. The *associated_data* is authenticated but is not included in the ciphertext. Because each message key is only used once, the AEAD nonce may handled in several ways: fixed to a constant; derived from *mk* alongside an independent AEAD encryption key; derived as an additional output from *KDF_CK()*; or chosen randomly and transmitted.
- ***DECRYPT(mk, ciphertext, associated_data)***: Returns the AEAD decryption of *ciphertext* with message key *mk*. If authentication fails, an exception will be raised that terminates processing.
- ***HEADER(dh_pair, pn, n)***: Creates a new message header containing the DH ratchet public key from the key pair in *dh_pair*, the previous chain length *pn*, and the message number *n*. The returned header object contains ratchet public key *dh* and integers *pn* and *n* and must ensure that *dh* is not *None*.
- ***CONCAT(ad, header)***: Encodes a message header into a parseable byte sequence, prepends the *ad* byte sequence, and returns the result. If *ad* is not guaranteed to be a parseable byte sequence, a length value should be prepended to the output to ensure that the output is parseable as a unique pair (*ad*, *header*).

A ***MAX_SKIP*** constant also needs to be defined. This specifies the maximum number of message keys that can be skipped in a single chain. It should be set high enough to tolerate routine lost or delayed messages, but low enough that a

malicious sender can't trigger excessive recipient computation.

3.2. State variables

The following state variables are tracked by each party:

- ***DHs***: DH Ratchet key pair (the "sending" or "self" ratchet key)
- ***DHr***: DH Ratchet public key (the "received" or "remote" key)
- ***RK***: 32-byte Root Key
- ***CKs, CKr***: 32-byte Chain Keys for sending and receiving
- ***Ns, Nr***: Message numbers for sending and receiving
- ***PN***: Number of messages in previous sending chain
- ***MKSKIPPED***: Dictionary of skipped-over message keys, indexed by ratchet public key and message number. Raises an exception if too many elements are stored.

In the Python code that follows, the state variables are accessed as members of a ***state*** object.

3.3. Initialization

Prior to initialization both parties must use some key agreement protocol to agree on a 32-byte shared secret key *SK* and Bob's ratchet public key. These values will be used to populate Alice's sending chain key and Bob's root key. Bob's chain keys and Alice's receiving chain key will be left empty, since they are populated by each party's first DH ratchet step.

(This assumes Alice begins sending messages first, and Bob doesn't send messages until he has received one of Alice's messages. To allow Bob to send messages immediately after initialization Bob's sending chain key and Alice's receiving chain key could be initialized to a shared secret. For the sake of simplicity we won't consider this further.)

Once Alice and Bob have agreed on *SK* and Bob's ratchet public key, Alice calls *RatchetInitAlice()* and Bob calls *RatchetInitBob()*:

```
def RatchetInitAlice(state, SK, bob_dh_public_key):
    state.DHs = GENERATE_DH()
    state.DHr = bob_dh_public_key
    state.RK, state.CKs = KDF_RK(SK, DH(state.DHs, state.DHr))
    state.CKr = None
    state.Ns = 0
    state.Nr = 0
    state.PN = 0
    state.MKSKIPPED = {}
```

```
def RatchetInitBob(state, SK, bob_dh_key_pair):
    state.DHs = bob_dh_key_pair
    state.DHr = None
    state.RK = SK
    state.CKs = None
    state.CKr = None
    state.Ns = 0
    state.Nr = 0
    state.PN = 0
    state.MKSKIPPED = {}
```

3.4. Encrypting messages

RatchetEncrypt() is called to encrypt messages. This function performs a symmetric-key ratchet step, then encrypts the message with the resulting message key. In addition to the message's *plaintext* it takes an AD byte sequence which is prepended to the header to form the associated data for the underlying AEAD encryption:

```
def RatchetSendKey(state):
    state.CKs, mk = KDF_CK(state.CKs)
    Ns = state.Ns
    state.Ns += 1
    return Ns, mk

def RatchetEncrypt(state, plaintext, AD):
    Ns, mk = RatchetSendKey(state)
    header = HEADER(state.DHs, state.PN, Ns)
    return header, ENCRYPT(mk, plaintext, CONCAT(AD, header))
```

3.5. Decrypting messages

RatchetDecrypt() is called to decrypt messages. This function does the following:

- If the message corresponds to a skipped message key this function decrypts the message, deletes the message key, and returns.
- Otherwise, if a new ratchet key has been received this function stores any skipped message keys from the receiving chain and performs a DH ratchet step to replace the sending and receiving chains.
- This function then stores any skipped message keys from the current receiving chain, performs a symmetric-key ratchet step to derive the relevant message key and next chain key, and decrypts the message.

If an exception is raised (e.g. message authentication failure) then the message is discarded and changes to the state object are discarded. Otherwise, the

decrypted plaintext is accepted and changes to the state object are stored:

```
def RatchetReceiveKey(state, header):
    mk = TrySkippedMessageKeys(state, header)
    if mk != None:
        return mk
    if header.dh != state.DHr:
        SkipMessageKeys(state, header.pn)
        DHRatchet(state, header)
    SkipMessageKeys(state, header.n)
    state.CKr, mk = KDF_CK(state.CKr)
    state.Nr += 1
    return mk

def RatchetDecrypt(state, header, ciphertext, AD):
    mk = RatchetReceiveKey(state, header)
    return DECRYPT(mk, ciphertext, CONCAT(AD, header))

def TrySkippedMessageKeys(state, header):
    if (header.dh, header.n) in state.MKSKIPPED:
        mk = state.MKSKIPPED[header.dh, header.n]
        del state.MKSKIPPED[header.dh, header.n]
        return mk
    else:
        return None

def SkipMessageKeys(state, until):
    if state.Nr + MAX_SKIP < until:
        raise Error()
    if state.CKr != None:
        while state.Nr < until:
            state.CKr, mk = KDF_CK(state.CKr)
            state.MKSKIPPED[state.DHr, state.Nr] = mk
            state.Nr += 1

def DHRatchet(state, header):
    state.PN = state.Ns
    state.Ns = 0
    state.Nr = 0
    state.DHr = header.dh
    state.RK, state.CKr = KDF_RK(state.RK, DH(state.DHs, state.DHr))
    state.DHs = GENERATE_DH()
    state.RK, state.CKs = KDF_RK(state.RK, DH(state.DHs, state.DHr))
```

4. Double Ratchet with header encryption

4.1. Overview

This section describes the **header encryption** variant of the Double Ratchet.

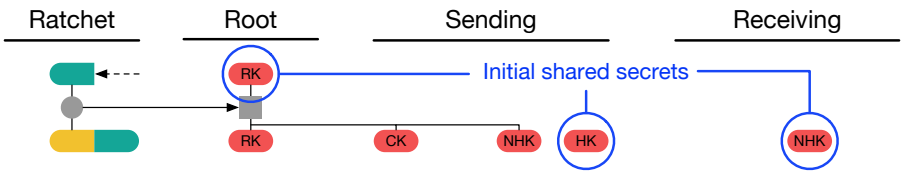
Message headers contain ratchet public keys and (PN, N) values. In some cases it may be desirable to encrypt the headers so that an eavesdropper can't tell which messages belong to which sessions, or the ordering of messages within a session.

With header encryption each party stores a symmetric **header key** and **next header key** for both the sending and receiving directions. The sending header key is used for encrypting headers for the current sending chain.

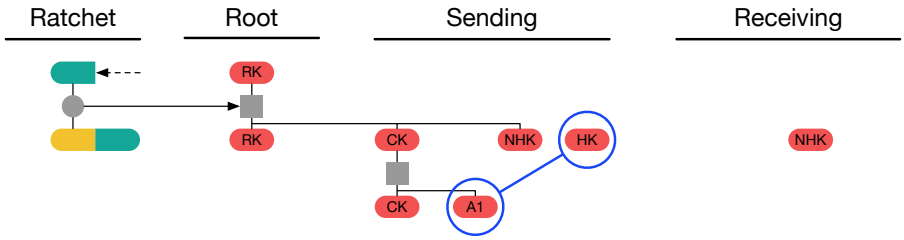
When a recipient receives a message she must first associate the message with its relevant Double Ratchet session (assuming she has different sessions with different parties). How this is done is outside of the scope of this document, although the Pond protocol offers some ideas [6].

After associating the message with a session, the recipient attempts to decrypt the header with that session's receiving header key, next header key, and any header keys corresponding to skipped messages. Successful decryption with the next header key indicates the recipient must perform a DH ratchet step. During a DH ratchet step the next header keys replace the current header keys, and new next header keys are taken as additional output from the root KDF.

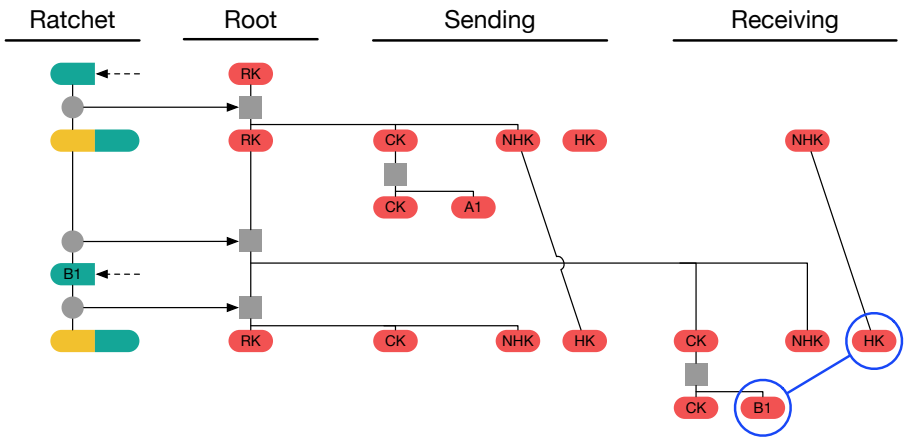
In the below diagram Alice has been initialized with Bob's ratchet public key and shared secrets for the initial root key, the sending header key (HK), and the receiving next header key (NHK). As part of initialization Alice generates her ratchet key pair and updates the root chain to derive a new root key, sending chain key, and sending next header key (NHK):



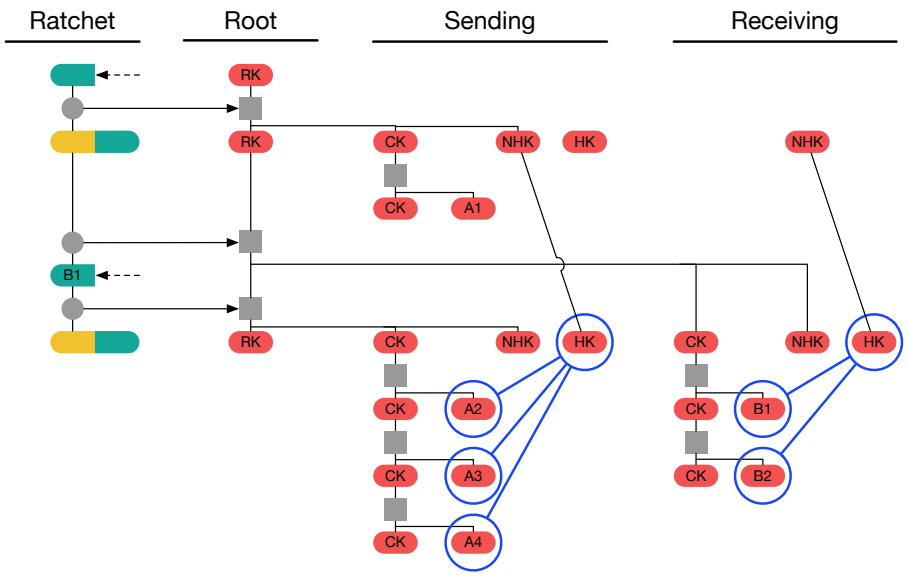
When Alice sends her first message *A1*, she encrypts its header with the sending header key she was initialized with:



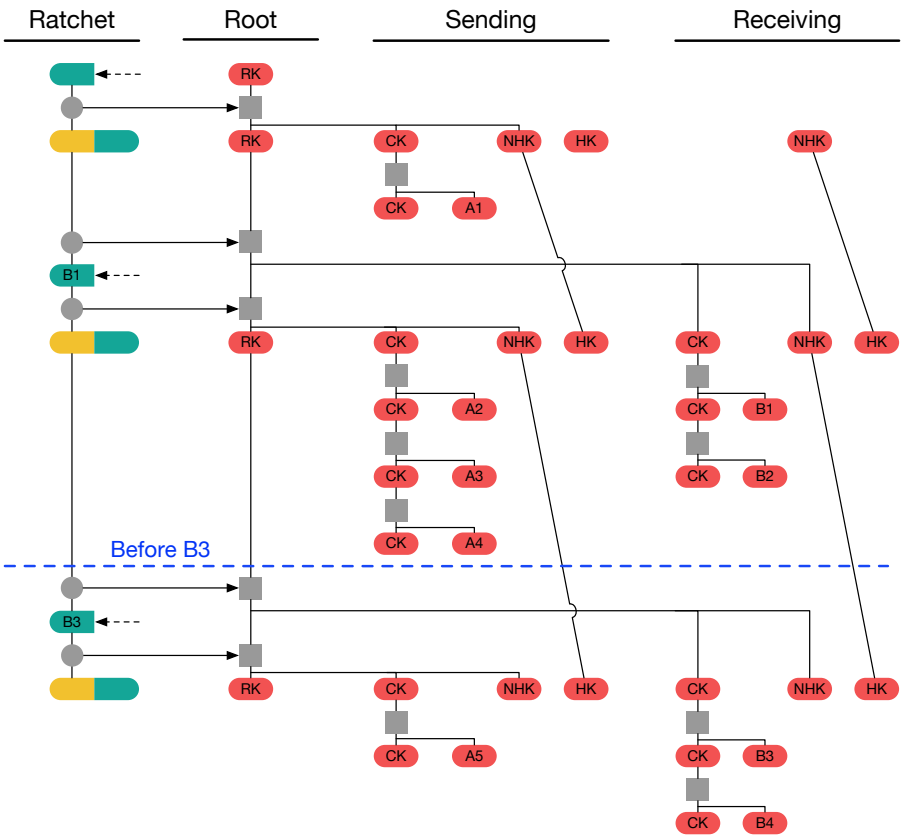
If Alice next receives a response B1 from Bob, its header will be encrypted with the receiving next header key that she was initialized with. Alice applies a DH ratchet step which shifts the next header keys into the current header keys, and generates new next header keys:



Alice next sends a message *A2*, then receives a message *B2* using the current receiving header key and containing the same ratchet public key she received in message *B1*. Alice then sends messages *A3* and *A4*. The current header keys are used for all sent and received messages:



Alice then receives message *B3* containing Bob's next ratchet key and with its header encrypted by the next receiving header key. Successful header decryption with the next header key will trigger a DH ratchet step. Alice then receives *B4* with the same ratchet key and header key, then sends a message *A5*. Alice's final state will be as follows:



4.2. External functions

Additional functions are required for header encryption:

- ***HENCRYPT(hk, plaintext)***: Returns the AEAD encryption of *plaintext* with header key *hk*. Because the same *hk* will be used repeatedly, the AEAD nonce must either be a stateful non-repeating value, or must be a random non-repeating value chosen with at least 128 bits of entropy.
- ***HDECRYPT(hk, ciphertext)***: Returns the authenticated decryption of *ciphertext* with header key *hk*. If authentication fails, or if the header key *hk* is empty (*None*), returns *None*.
- ***KDF_RK_HE(rk, dh_out)***: Returns a new root key, chain key, and next header key as the output of applying a KDF keyed by root key *rk* to a Diffie-Hellman output *dh_out*.

4.3. State variables

Additional state variables are required:

- ***HKs, HKr***: 32-byte Header Keys for sending and receiving
- ***NHKs, NHKr***: 32-byte Next Header Keys for sending and receiving

The following variable's definition is changed:

- ***MKSKIPPED***: Dictionary of skipped-over message keys, indexed by header key and message number. Raises an exception if too many elements are stored.

4.4. Initialization

Some additional shared secrets must be used to initialize the header keys:

- Alice's sending header key and Bob's next receiving header key must be set to the same value, so that Alice's first message triggers a DH ratchet step for Bob.
- Alice's next receiving header key and Bob's next sending header key must be set to the same value, so that after Bob's first DH ratchet step, Bob's next message triggers a DH ratchet step for Alice.

Once Alice and Bob have agreed on *SK*, Bob's ratchet public key, and these additional values, Alice calls *RatchetInitAliceHE()* and Bob calls *RatchetInitBobHE()*:

```
def RatchetInitAliceHE(state, SK, bob_dh_public_key, shared_hka, shared_nhkb):
    state.DHRs = GENERATE_DH()
    state.DHRr = bob_dh_public_key
    state.RK, state.CKs, state.NHKs = KDF_RK_HE(SK, DH(state.DHRs, state.DHRr))
    state.CKr = None
    state.Ns = 0
    state.Nr = 0
    state.PN = 0
    state.MKSKIPPED = {}
    state.HKs = shared_hka
    state.HKr = None
    state.NHKr = shared_nhkb

def RatchetInitBobHE(state, SK, bob_dh_key_pair, shared_hka, shared_nhkb):
    state.DHRs = bob_dh_key_pair
    state.DHRr = None
    state.RK = SK
    state.CKs = None
    state.CKr = None
    state.Ns = 0
    state.Nr = 0
    state.PN = 0
    state.MKSKIPPED = {}
    state.HKs = None
    state.NHKs = shared_nhkb
    state.HKr = None
    state.NHKr = shared_hka
```

4.5. Encrypting messages

The *RatchetEncryptHE()* function is called to encrypt messages with header encryption:

```
def RatchetEncryptHE(state, plaintext, AD):
    state.CKs, mk = KDF_CK(state.CKs)
    header = HEADER(state.DHRs, state.PN, state.Ns)
    enc_header = HENCRYPT(state.HKs, header)
    state.Ns += 1
    return enc_header, ENCRYPT(mk, plaintext, CONCAT(AD, enc_header))
```

4.6. Decrypting messages

RatchetDecryptHE() is called to decrypt messages with header encryption:

```
def RatchetDecryptHE(state, enc_header, ciphertext, AD):
    plaintext = TrySkippedMessageKeysHE(state, enc_header, ciphertext, AD)
    if plaintext != None:
        return plaintext
    header, dh_ratchet = DecryptHeader(state, enc_header)
    if dh_ratchet:
        SkipMessageKeysHE(state, header.pn)
        DHRatchetHE(state, header)
    SkipMessageKeysHE(state, header.n)
    state.CKr, mk = KDF_CK(state.CKr)
    state.Nr += 1
    return DECRYPT(mk, ciphertext, CONCAT(AD, enc_header))

def TrySkippedMessageKeysHE(state, enc_header, ciphertext, AD):
    for ((hk, n), mk) in state.MKSKIPPED.items():
        header = HDECRYPT(hk, enc_header)
        if header != None and header.n == n:
            del state.MKSKIPPED[hk, n]
            return DECRYPT(mk, ciphertext, CONCAT(AD, enc_header))
    return None

def DecryptHeader(state, enc_header):
    header = HDECRYPT(state.HKr, enc_header)
    if header != None:
        return header, False
    header = HDECRYPT(state.NHKr, enc_header)
    if header != None:
        return header, True
    raise Error()

def SkipMessageKeysHE(state, until):
    if state.Nr + MAX_SKIP < until:
        raise Error()
    if state.CKr != None:
        while state.Nr < until:
            state.CKr, mk = KDF_CK(state.CKr)
            state.MKSKIPPED[state.HKr, state.Nr] = mk
            state.Nr += 1
```

```
def DHRatchetHE(state, header):
    state.PN = state.Ns
    state.Ns = 0
    state.Nr = 0
    state.HKs = state.NHKs
    state.HKr = state.NHKr
    state.DHRr = header.dh
    state.RK, state.CKr, state.NHKr = KDF_RK_HE(state.RK, DH(state.DHRs, state.DHRr))
    state.DHRs = GENERATE_DH()
    state.RK, state.CKs, state.NHKs = KDF_RK_HE(state.RK, DH(state.DHRs, state.DHRr))
```

5. The Sparse Post-Quantum Ratchet

This section describes the *Sparse Post-Quantum Ratchet protocol*, a secure messaging protocol built on a Sparse Continuous Key Agreement (SCKA) protocol.

The Double Ratchet protocol described above does not provide security against attacks using cryptographically relevant quantum computers. For applications without bandwidth constraints, [7] shows a way to generalize the Double Ratchet protocol presented above to use what is called a *Continuous Key Agreement* (CKA) protocol. Informally, a CKA is a protocol that sends messages back and forth between Alice and Bob in a way that a new shared key is generated with each message. For example, the Double Ratchet protocol from Section 3 corresponds to a CKA where parties send fresh ephemeral keys with each message, and the output key after any round is the Diffie-Hellman key agreement the ephemeral keys exchanged in the previous two rounds (which can be efficiently computed by both Alice and Bob). Moreover, it is easy to construct post-quantum secure CKA protocols — and hence get a post-quantum secure variant of the Double Ratchet protocol — using any post-quantum secure Key Encapsulation Mechanism (KEM), such as ML-KEM. While this is elegant, existing post-quantum KEMs require exchanging messages that are orders of magnitude larger than elliptic curve keys, which can be impractical for bandwidth-limited applications.

To address this issue and enable robust, quantum-safe ratcheting protocols in a bandwidth-limited environment, Sparse Continuous Key Agreement (SCKA) protocols have been introduced by [8]. Just like CKA protocols, SCKA protocols allow two parties to regularly produce shared secrets. However, one drops the requirement that fresh shared keys are produced with each message, as well as insistence that the parties alternate their messages. This allows one to design natural, post-quantum secure SCKA protocols in an environment with bandwidth limits [8], as one no longer needs to transmit full post-quantum key exchange messages in every single application message. We describe the syntax and intuitive security of SCKA in Section 5.1, leaving the actual design of a suitable post-quantum SCKA (called the ML-KEM Braid protocol) in a

companion document [9].

In the remainder of Section 5 we present the details of *Sparse Post-Quantum Ratchet*, a secure messaging protocol built generically from any SCKA protocol. Intuitively, the generalized protocol will only create new sender and receiver chains when the SCKA produces a new secret. As a subtlety, it will need to create both a sender chain and a receiver chain with this new secret, as both parties might exchange application messages during the time in between successive SCKA secret generations. Moreover, in addition to post-quantum security and good bandwidth usage, it will inherit security properties similar to the original Double Ratchet protocol. Intuitively, when using the ML-KEM Braid as the SCKA protocol, this Sparse Post-Quantum Ratchet gains post-quantum security, but more messages must be exchanged to attain post-compromise security.

5.1 Sparse Continuous Key Agreement

SCKA is defined formally in [8] and a high-level description is available in [9].

An SCKA protocol provides two functions that output shared secrets. These shared secrets are identified by ordered "epoch identifiers", and the SCKA functions output epoch information needed to use these secrets correctly:

- **SCKAInitAlice(sk) → state**: Initialize Alice's state using an initial shared secret *sk*.
- **SCKAInitBob(sk) → state**: Initialize Bob's state using an initial shared secret *sk*.
- **SCKASend(state) → (msg, sending_epoch, output_key)**
  - **msg**: Opaque message data for the SCKA protocol
  - **sending_epoch**: Latest epoch guaranteed known by receiver after processing *msg*
  - **output_key**: Either None or *(key_epoch, key)* where *key_epoch* identifies the epoch of *key*
- **SCKAReceive(state, msg) → (receiving_epoch, output_key)**:
  - **msg**: Opaque message data for the SCKA protocol
  - **receiving_epoch**: Epoch emitted as *sending_epoch* when sender generated *msg*
  - **output_key**: Either None or *(key_epoch, key)* where *key_epoch* identifies the epoch of *key*

The Sparse Post-Quantum Ratchet is a natural adaptation of the Elliptic Curve Double Ratchet to the SCKA setting. Instead of sending Elliptic Curve public keys with each application message, we send the message output by a call to *SCKASend()*. Whenever *SCKASend()* or *SCKAReceive()* return a non-null *key*, we mix this with our root key to create a new root key as well as a new sender KDF chain and receiver KDF chain for this new epoch.

When encrypting a message we use the *sending_epoch* returned by *SCKASend()* to look up the correct sending chain to use and we compute a message key from

that KDF chain. The SCKA protocol guarantees that the *sending_epoch* will be the latest epoch known by the sender that will also be known by the receiver after processing *msg*.

Similarly, when receiving a message, we use the *receiving_epoch* returned by *SCKAReceive()* to look up the needed receiver chain, and retrieve or compute a message key from that KDF chain. The SCKA protocol guarantees that the receiving party will know the secret for *receiving_epoch* once *msg* is processed and that the call to *SCKASend()* that generated *msg* output *sending_epoch* with *sending_epoch == receiving_epoch*.

5.2 External functions

- ***KDF_SCKA_INIT(sk)***: Returns a triple (32-byte root key, 32-byte sender chain key, 32-byte receiver chain key) as the output of applying a KDF keyed by a 32-byte secret *sk* to some unique constant specifying the protocol in use and its parameters.
- ***KDF_SCKA_RK(rk, scka_output)***: Returns a triple (32-byte root key, 32-byte sender chain key, 32-byte receiver chain key) as the output of applying a KDF keyed by a 32-byte root key *rk* to a key output by the SCKA protocol.
- ***KDF_SCKA_CK(ck, ctr)***: Returns a pair (32-byte chain key, 32-byte message key) as the output of applying a KDF keyed by a 32-byte chain key *ck* to *ctr* concatenated with some unique constant specifying the protocol in use and its parameters.
- ***KDFChain(ck)***: Initializes a KDF Chain with chain key $CK = ck$ and counter $N = 0$.

5.3 State variables

- **RK**: a 32-byte root key.
- **epoch**: The latest epoch for which SCKA keys have been incorporated.
- **kdfchains**: a table indexed by epoch with values containing two KDF chains: *send* and *receive*.
- **MKSKIPPED**: a map from epoch to a map of message numbers to skipped message keys.
- **direction**: A flag indicating the role the participant plays in the protocol, one of *A2B* or *B2A*.
- **scka_state**: the opaque state of an SCKA protocol.

5.4 Initialization

Prior to initialization both parties must use some key agreement protocol to agree on a 32-byte shared secret key *SK*. This value will be used to populate

Alice and Bob's initial RK and KDF chains, as well as to initialize the SCKA state.

```
def RatchetInitAliceSCKA(state, SK):
  state.scka_state = SCKAInitAlice(SK)
  state.direction = A2B
  state.RK, CKs, CKr = KDF_SCKA_INIT(SK)
  state.epoch = 0
  state.kdfchains = {}
  state.kdfchains[0] = {}
  state.MKSKIPPED = {}
  state.kdfchains[0].send = KDFChain(CKs)
  state.kdfchains[0].receive = KDFChain(CKr)

def RatchetInitBobSCKA(state, SK):
  state.scka_state = SCKAInitBob(SK)
  state.direction = B2A
  state.RK, CKr, CKs = KDF_SCKA_INIT(SK) # note the reordering of CKs and CKr
  state.epoch = 0
  state.kdfchains = {}
  state.kdfchains[0] = {}
  state.MKSKIPPED = {}
  state.kdfchains[0].send = KDFChain(CKs)
  state.kdfchains[0].receive = KDFChain(CKr)
```

5.5 Encrypting Messages

Since the SCKA may output a new shared secret when sending a message, the protocol may need to advance the public ratchet when computing the message key used to encrypt an outgoing message. Otherwise, Sparse Post-Quantum Ratchet encryption is similar to Double Ratchet encryption.

```
def SCKARatchetSendKey(state):
    msg, sending_epoch, output_key = SCKASend(state.scka_state)
    if output_key != None:
        key_epoch, key = output_key
        # Advance to new epoch
        assert state.epoch + 1 == key_epoch
        state.RK, CKs, CKr = KDF_SCKA_RK(state.RK, key)
        if state.direction == B2A:
            (CKs, CKr) = (CKr, CKs)

        # Create new chains
        state.kdfchains[key_epoch] = {
            'send': KDFChain(CKs),
            'receive': KDFChain(CKr)
        }
```

```
        state.epoch = key_epoch
        ClearOldEpochs(state, sending_epoch)

    # Continue with message key derivation
    state.kdfchains[sending_epoch - 1].send = None
    state.kdfchains[sending_epoch].send.N += 1
    state.kdfchains[sending_epoch].send.CK, mk
        = KDF_SCKA_CK(state.kdfchains[sending_epoch].send.CK, state.kdfchains[sending_epoch]
    return msg, state.kdfchains[sending_epoch].send.N, mk

def SCKARatchetEncrypt(state, plaintext, AD):
    msg, n, mk = SCKARatchetSendKey(state)
    header = SCKA_HEADER(msg, n)
    return header, ENCRYPT(mk, plaintext, CONCAT(AD, header))

def ClearOldEpochs(state, sending_epoch):
    state.kdfchains[sending_epoch - 2] = None
    state.MKSKIPPED[sending_epoch - 2] = None
```

5.6 Decrypting Messages

When receiving a message, the SCKA protocol takes the place of *DHRatchet()* in the Double Ratchet. When the SCKA emits a new shared key, it is used to advance the root key and create sender and receiver KDF chains for the new epoch.

```
def SCKARatchetReceiveKey(state, header):
    receiving_epoch, output_key = SCKAReceive(state.scka_state, header.msg)
    if output_key != None:
        key_epoch, key = output_key
        assert state.epoch + 1 == key_epoch
        state.RK, CKs, CKr = KDF_SCKA_RK(state.RK, key)
        if state.direction == B2A:
            (CKs, CKr) = (CKr, CKs)

        # Create new chains
        state.kdfchains[key_epoch] = {
            'send': KDFChain(CKs),
            'receive': KDFChain(CKr)
        }
        state.epoch = key_epoch

    # Continue with message key derivation
    mk = TrySkippedMessageKeys(state, receiving_epoch, header.n)
    if mk != None:
        return mk
```

```
    SkipMessageKeys(state, receiving_epoch, header.n)
    state.kdfchains[receiving_epoch].receive.N += 1
    state.kdfchains[receiving_epoch].receive.CK, mk
        = KDF_SCKA_CK(
              state.kdfchains[receiving_epoch].receive.CK,
              state.kdfchains[receiving_epoch].receive.N)
    return mk

def SCKARatchetDecrypt(state, header, ciphertext, AD):
    mk = SCKARatchetReceiveKey(state, header)
    return DECRYPT(mk, ciphertext, CONCAT(AD, header))

def TrySkippedMessageKeys(state, key_epoch, n):
    if key_epoch in state.MKSKIPPED and n in state.MKSKIPPED[key_epoch]:
        mk = state.MKSKIPPED[key_epoch][n]
        del state.MKSKIPPED[key_epoch][n]
        if len(state.MKSKIPPED[key_epoch]) == 0:
            del state.MKSKIPPED[key_epoch]
        return mk
    else:
        return None

def SkipMessageKeys(state, epoch, until):
    if state.kdfchains[epoch].receive == None:
        return
    if state.kdfchains[epoch].receive.N + MAX_SKIP < until:
        raise Error()
    while state.kdfchains[epoch].receive.N < until:
        state.kdfchains[epoch].receive.N += 1
        state.kdfchains[epoch].receive.CK, mk
            = KDF_SCKA_CK(state.kdfchains[epoch].receive.CK, state.kdfchains[epoch].receive.
        if epoch not in state.MKSKIPPED:
            state.MKSKIPPED[epoch] = {}
        state.MKSKIPPED[epoch][state.kdfchains[epoch].receive.N] = mk
```

5.7 Clearing past epoch state

The Sparse Post-Quantum Ratchet clears old KDF Chain state by calling the function *ClearOldEpochs(state, epoch)* to ensure that the state only stores the KDF Chains for two epochs at any time. Note that this function *also* clears the keys stored in *MKSKIPPED* for older epochs, limiting a receiver's ability to decrypt old out of order messages. This is intended for deployments where many messages may be dropped (e.g. ephemeral messages like typing indicators) but messages are rarely delivered far out of order.

Implementors may also clear the state of past KDF Chains by communicating

the number of messages sent in the previous epoch, as is done with the Double Ratchet. Specifically they would add the following variables to their state:

- **sending_epoch**: The epoch currently being used to send messages.
- **receiving_epoch**: The latest epoch currently being used to receive messages.
- **PN**: The number of messages sent in the last sending chain, added to the header of every sent message.

They implement the following functions:

```
def SealPrevSendingChain(state):
    state.PN = state.kdfchains[state.sending_epoch].send.N
    state.kdfchains[state.sending_epoch].send = None

def SealPrevReceivingChain(state, PN):
    SkipMessageKeys(state, state.receiving_epoch, PN)
    state.kdfchains[state.receiving_epoch].receive = None
```

These functions can then be used to clear old state when an epoch advances. Message sending is modified as follows:

```
def SCKARatchetSendKey(state):
    msg, sending_epoch, key_epoch, key = SCKASend(state.scka_state)
    if sending_epoch > state.sending_epoch:
        assert sending_epoch == state.sending_epoch + 1
        SealPrevSendingChain(state)
        state.sending_epoch = sending_epoch
    # remainder of the function is unchanged
    # ...

def SCKARatchetEncrypt(state, plaintext, AD):
    msg, PN, n, mk = SCKARatchetSendKey(state)
    header = SCKA_HEADER(msg, PN, n)
    return header, ENCRYPT(mk, plaintext, CONCAT(AD, header))
```

Message receipt is modified similarly:

```
def SCKARatchetReceiveKey(state, header):
    receiving_epoch, key_epoch, key = SCKAReceive(state.scka_state, header.msg)
    if receiving_epoch > state.receiving_epoch:
        assert receiving_epoch == state.receiving_epoch + 1
        SealPrevReceivingChain(state, header.PN)
        state.receiving_epoch = receiving_epoch
    # remainder of the function is unchanged.
```

If this approach is used in place of using *ClearOldEpochs(state, epoch)*, implementors should find some other means to remove old keys from the *MKSKIPPED* data structure so it does not grow unboundedly.

6. The Triple Ratchet: combining Secure Messaging protocols for hybrid security

This section describes the *Triple Ratchet* protocol.

The Double Ratchet provides security guarantees as long as certain Diffie-Hellman assumptions are hard [7], [10], [11]. Similarly, a Sparse Post-Quantum Ratchet based on a quantum-safe SCKA protocol provides post-quantum security guarantees [8]. In this section we describe how to combine these into a protocol that provides hybrid security: an attacker must break both the elliptic curve and the post-quantum assumptions in order to break the security guarantees of the protocol.

6.1 Overview

The hybridization strategy is simple: run a Double Ratchet and a Sparse Post-Quantum Ratchet in parallel, but only use them to produce *message keys*, not to perform full encryption or decryption of messages. Now, whenever an encryption key is needed, call each of these Secure Messaging protocols to obtain message keys and then use a KDF to combine these two message keys into an encryption key.

6.2 State variables

A Triple Ratchet state consists of a Double Ratchet state and a Sparse Post-Quantum Ratchet state:

- ***ec_state***: the state of an Elliptic Curve Double Ratchet.
- ***spqr_state***: the state of a Sparse Post-Quantum Ratchet.

6.3 External functions

- ***KDF_HYBRID(ecmk, pqmk)***: Returns an AEAD encryption key as the output of applying a KDF keyed by the concatenation of two 32-byte secrets to some unique constant specifying the protocol in use and its parameters.
- ***SCKA_HEADER(scka_msg, pn)***: Creates a new message header containing a message output by the SCKA protocol underlying SPQR and the previous chain length *pn*.

6.4 Initialization

Prior to initialization both parties must use some key agreement protocol to agree on 32-byte shared secret keys *SKec* and *SKscka*, and Bob's ratchet public key. These values will be used to initialize the Double Ratchet and the Sparse Post-Quantum Ratchet.

```
def RatchetInitAliceTR(state, SKec, SKscka, bob_dh_public_key):
    RatchetInitAlice(state.ec_state, SKec, bob_dh_public_key)
    RatchetInitSCKA(state.spqr_state, SKscka)

def RatchetInitBobTR(state, SKec, SKscka, bob_dh_key_pair):
    RatchetInitBob(state.ec_state, SKec, bob_dh_key_pair)
    RatchetInitSCKA(state.spqr_state, SKscka)
```

6.5 Encrypting Messages

Triple Ratchet messages use a composite header containing:

- **ec_header**: Standard Double Ratchet header (dh, pn, n)
- **scka_header**: SCKA message data and metadata (msg, n)

The header structure must ensure both components can be parsed unambiguously.

To encrypt a message, get the message keys from both ratchets along with needed header information, then securely combine the two message keys to produce the encryption key.

```
def TripleRatchetEncrypt(state, plaintext, AD):
    ecNs, ec_mk = RatchetSendKey(state.ec_state)
    scka_msg, pqN, pq_mk = SCKARatchetSendKey(state.spqr_state)
    mk = KDF_HYBRID(ec_mk, pq_mk)
    header.ec_header = HEADER(state.ec_state.DHs, state.ec_state.PN, ecNs)
    header.scka_header = SCKA_HEADER(scka_msg, pqN)
    return header, ENCRYPT(mk, plaintext, CONCAT(AD, header))
```

6.6 Decrypting Messages

```
def TripleRatchetDecrypt(state, header, ciphertext, AD):
    ec_mk = RatchetReceiveKey(state.ec_state, header.ec_header)
    pq_mk = SCKARatchetReceiveKey(state.spqr_state, header.scka_header)
    mk = KDF_HYBRID(ec_mk, pq_mk)
    return DECRYPT(mk, ciphertext, CONCAT(AD, header))
```

7. Implementation considerations

7.1. Integration with PQXDH

All ratchet protocols described in this document - the Double Ratchet, Sparse Post-Quantum Ratchet, and the Triple Ratchet - can be used in combination with the PQXDH key agreement protocol [1]. The ratchet protocol plays the role of a "post-PQXDH" protocol which takes the session key SK negotiated by PQXDH and uses it to derive its initial root key(s).

The following outputs from PQXDH are used by the ratchet protocols:

- The SK output from PQXDH becomes the SK input to Double Ratchet initialization (see Section 3.3) or Sparse Post-Quantum Ratchet initialization (see Section 5.4). For the Triple Ratchet the SK output from PQXDH should be expanded into two 32-byte keys $SKec$ and $SKscka$ using a key derivation function and used as input for the Triple Ratchet initialization (see Section 6.4).
- The AD output from PQXDH becomes the AD input to ratchet protocol encryption and decryption (see Section 3.4 and Section 3.5).
- Bob's signed prekey from PQXDH (SPK_B) becomes Bob's initial ratchet public key (and corresponding key pair) for Double Ratchet initialization.

Any message encrypted using Alice's initial ratchet protocol state before any messages have been received can serve as an "initial ciphertext" for PQXDH. To deal with the possibility of lost or out-of-order messages, a recommended pattern is for Alice to repeatedly send the same PQXDH initial message prepended to all of her ratchet protocol messages until she receives Bob's first ratchet protocol response message.

7.2. Recommended cryptographic algorithms

The following choices are recommended for instantiating the cryptographic functions from Section 3.1, Section 5.2, and Section 6.3. Functions used for the Sparse Post-Quantum Ratchet and the Triple Ratchet respectively depend on constant strings *SPQR_PROTOCOL_INFO* and *TR_PROTOCOL_INFO* that should uniquely identify the protocol and version in use.

- ***GENERATE_DH()***: This function is recommended to generate a key pair based on the Curve25519 or Curve448 elliptic curves [12].
- ***DH(dh_pair, dh_pub)***: This function is recommended to return the output from the X25519 or X448 function as defined in [12]. There is no need to check for invalid public keys.
- ***KDF_RK(rk, dh_out)***: This function is recommended to be implemented using HKDF [3] with SHA-256 or SHA-512 [13] and: using *rk* as HKDF *salt*, *dh_out* as HKDF *input key material*, and an application-specific byte sequence as HKDF *info*. The *info* value should be chosen to be distinct from other uses of HKDF in the application.
- ***KDF_CK(ck)***: HMAC [2] with SHA-256 or SHA-512 [13] is recommended, using *ck* as the HMAC key and using separate constants as input (e.g. a single byte 0x01 as input to produce the message key, and a single byte 0x02 as input to produce the next chain key). If *ck* is *None* this function must fail in a way that terminates processing.
- ***ENCRYPT(mk, plaintext, associated_data)***: This function is recommended to be implemented with an AEAD encryption scheme based on either SIV or a composition of CBC with HMAC [5], [14]. These schemes provide some misuse-resistance in case a key is mistakenly used multiple times. A concrete recommendation based on CBC and HMAC is as follows:
  - HKDF is used with SHA-256 or SHA-512 to generate 80 bytes of output. The HKDF *salt* is set to a zero-filled byte sequence equal to the hash's output length. HKDF *input key material* is set to *mk*. HKDF *info* is set to an application-specific byte sequence distinct from other uses of HKDF in the application.
    - The HKDF output is divided into a 32-byte encryption key, a 32-byte authentication key, and a 16-byte IV.
    - The plaintext is encrypted using AES-256 in CBC mode with PKCS#7 padding, using the encryption key and IV from the previous step [15], [16].
    - HMAC is calculated using the authentication key and the same hash function as above [2]. The HMAC input is the *associated_data* prepended to the ciphertext. The HMAC output is appended to the ciphertext.

- ***SCKA***: For hybrid post-quantum security, this protocol is recommended to be the ML-KEM Braid protocol described in [9].
- ***KDF_SCKA_INIT(sk)*** This function is recommended to be implemented using HKDF [3] with SHA-256 or SHA-512 [13] and:
  - **salt**: zero-filled 32 bytes
  - **ikm**: *sk*
  - **info**: Some unique constant specifying the protocol and version, followed by "Chain Start"
  - **length**: 96
- ***KDF_SCKA_RK(rk, scka_out)***: This function is recommended to be implemented using HKDF [3] with SHA-256 or SHA-512 [13] and:
  - **salt**: *rk*
  - **ikm**: *scka_output*
  - **info**: *SPQR_PROTOCOL_INFO* followed by "Chain Add Epoch"
  - **length**: 96
- ***KDF_SCKA_CK(ck, ctr)***: This function is recommended to be implemented using HKDF [3] with SHA-256 or SHA-512 [13] and:
  - **salt**: zero-filled 32 bytes
  - **ikm**: *sk*
  - **info**: *SPQR_PROTOCOL_INFO* followed by "Chain Start"
  - **length**: 64
- ***KDF_HYBRID(ec_mk, scka_mk)***: This function is recommended to be implemented using HKDF [3] with SHA-256 or SHA-512 [13] and:
  - **salt**: *scka_mk*
  - **ikm**: *ec_mk*
  - **info**: *TR_PROTOCOL_INFO*
  - **length**: Key length required by *AEAD*.

8. Security considerations

8.1. Secure deletion

These ratchet protocols are designed to provide security against an attacker who records encrypted messages and then compromises the sender or receiver at a later time. This security could be defeated if deleted plaintext or keys could be recovered by an attacker with low-level access to the compromised device. Recovering deleted data from storage media is a complicated topic which is outside the scope of this document.

8.2. Recovery from compromise

These ratchet protocols are designed to recover security against a passive eavesdropper who observes encrypted messages after compromising one (or both) of the parties to a session. Despite this mitigation, a compromise of secret keys or of device integrity will have a devastating effect on the security of future communications. For example:

- The attacker could use the compromised keys to impersonate the compromised party (e.g. using the compromised party's identity private key with PQXDH to create new sessions).
- The attacker could substitute her own ratchet keys via continuous active man-in-the-middle attack, to maintain eavesdropping on the compromised session.
- The attacker could modify a compromised party's RNG so that future ratchet private keys are predictable.

If a party suspects its keys or devices have been compromised, it must replace them immediately.

8.3. Cryptanalysis and ratchet public keys

Because all DH ratchet computations and SCKA secrets are mixed into the root key, an attacker who can decrypt a session with passive cryptanalysis might lose this ability if she fails to observe some ratchet public key.

This is not a reliable countermeasure against cryptanalysis, of course. If weaknesses are discovered in any of the cryptographic algorithms a session relies upon, the session should be discarded and replaced with a new session using strong cryptography.

8.4. Deletion of skipped message keys

Storing skipped message keys introduces some risks:

- A malicious sender could induce recipients to store large numbers of skipped message keys, possibly causing denial-of-service due to consuming storage space.
- The lost messages may have been seen (and recorded) by an attacker, even though they didn't reach the recipient. The attacker can compromise the intended recipient at a later time to retrieve the skipped message keys.

To mitigate the first risk parties should set reasonable per-session limits on the number of skipped message keys that will be stored (e.g. 1000). To mitigate the second risk parties should delete skipped message keys after an appropriate interval. Deletion could be triggered by a timer, or by counting a number of events (messages received, DH ratchet steps, etc.).

8.5. Deferring new ratchet key generation

During each DH ratchet step a new ratchet key pair and sending chain are generated. As the sending chain is not needed right away, these steps could be deferred until the party is about to send a new message. This would slightly increase security by shortening the lifetime of ratchet keys, at the cost of some complexity.

8.6. Truncating authentication tags

If the *ENCRYPT()* function is implemented using CBC and HMAC as described in Section 7.2, then truncating the final HMAC output to 128 bits to reduce message size is acceptable. Truncating it further might be acceptable, though requires careful analysis. In no case should the final HMAC be truncated to less than 64 bits.

If the *ENCRYPT()* function is implemented differently, then truncation might require a more complicated analysis and is not recommended.

8.7. Implementation fingerprinting

If this protocol is used in settings with anonymous parties, care should be taken that implementations behave identically in all cases.

In an anonymous context, implementations are advised to follow the algorithms from Sections 3 and 4 precisely. Such implementations are also advised to use identical limits for the number of skipped message keys stored, and identical deletion policies for skipped message keys. Deletion policies should be based on deterministic events (e.g. messages received), rather than time.

8.8 Choice of SCKA protocol

The Triple Ratchet described in Section 6 was proposed in [17] where it is proven to provide *hybrid* security. It does not weaken the security guarantees of the Double Ratchet and it adds PCS and FS guarantees provided by the SCKA. The ML-KEM Braid Protocol [9], for example, provides FS and will provide Module LWE-based PCS healing for messages after two SCKA output keys are mixed into the ratchet.

As discussed in [8], the PCS of SCKA protocols should be compared by looking at the number of messages in the "Vulnerable Message set" - the set of messages exposed to an attacker who compromises a user device. This can vary considerably depending on messaging behavior and on the way the SCKA protocol generates and manages secrets internally. We recommend the ML-KEM Braid protocol because it uses a standardized KEM and it has a small Vulnerable Message Set across a wide range of message sending behaviors. Applications with very specific message sending patterns - e.g. applications where devices are rarely

online at the same time and send large numbers of messages without receiving responses - may find that a different SCKA protocol provides better security.

8.9 Effect of dropped messages on PCS

These ratchet protocols are all designed for *immediate decryption*: they can tolerate dropped or out of order messages. Note, though, that for the Sparse Post-Quantum Ratchet dropped messages may slow down the progress of the public ratchet and hence degrade PCS. This does not happen with the classical Double Ratchet.

8.10 Deletion of old KDF Chain state

The Sparse Post-Quantum Ratchet offered an alternative mechanism for clearing past KDF Chain state in Section 5.7. Using this alternative does provide a security benefit, since it deletes sending chain keys as soon as they are not needed and it deletes receiving chain keys as soon as a message is received in a later epoch. An example of the security consequences of this difference can be found in Appendix A.1 of [18].

8.11 Harvest Now, Decrypt Later Attacks

The primary motivation for introducing post-quantum ratcheting today is protection against adversaries who compromise devices today, record encrypted communications, and wait for cryptographically relevant quantum computers. Against such adversaries, classical Double Ratchet healing provides no protection—all recorded messages remain vulnerable regardless of subsequent Diffie-Hellman based key rotation.

9. IPR

10. Acknowledgements

The Double Ratchet algorithm was designed by Trevor Perrin and Moxie Marlinspike.

The concept of a Diffie-Hellman ratchet was taken from the OTR protocol by Nikita Borisov, Ian Goldberg, and Eric Brewer [19].

Symmetric-key ratcheting is an old idea [20], [21]. It's been used in recent protocols like SCIMP and MinimaLT [22]–[24].

The term "ratchet" for forward-secure key updating was introduced by Adam Langley in Pond [25].

Thanks to Michael Rogers and Adam Back for mailing list discussions [26].

Thanks to Adam Langley for discussion on improving the receiving algorithm.

The security of this protocol and similar protocols has been analyzed by Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila [10], [11].

Thanks to Tom Ritter, Joseph Bonneau, Ximin Luo, Yan Zhu, Samuel Neves, Raphael Arias, and David J. Wu for editorial feedback.

Post-Quantum Extensions

The post-quantum extensions - the Triple Ratchet, SPQR, and the ML-KEM Braid - were designed by Graeme Connell and Rolfe Schmidt.

The construction of secure messaging from a Sparse Continuous Key Agreement protocol was introduced and analyzed in [8]. The hybridization technique used to construct the Triple Ratchet was introduced and analyzed in [17]. The authors of these papers, Benedikt Auerbach, Yevgeniy Dodis, Daniel Jost, Shuichi Katsumata, and Thomas Prest 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 SPQR using ProVerif, and 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.

9. References

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

[2] H. Krawczyk, M. Bellare, and R. Canetti, “HMAC: Keyed-Hashing for Message Authentication.” Internet Engineering Task Force; RFC 2104 (Informational); IETF, Feb-1997. http://www.ietf.org/rfc/rfc2104.txt

[3] 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

[4] B. Barak and S. Halevi, “A model and architecture for pseudo-random generation with applications to /dev/random.” Cryptology ePrint Archive, Report 2005/029, 2005. http://eprint.iacr.org/2005/029

[5] P. Rogaway, “Authenticated-encryption with Associated-data,” in Proceedings of the 9th ACM Conference on Computer and Communications Security, 2002. http://web.cs.ucdavis.edu/~rogaway/papers/ad.pdf

[6] A. Langley, “Pond,” 2012. https://github.com/agl/pond

[7] J. Alwen, S. Coretti, and Y. Dodis, “The double ratchet: Security notions, proofs, and modularization for the signal protocol,” in Advances in cryptology - EUROCRYPT 2019 - 38th annual international conference on the theory and applications of cryptographic techniques, darmstadt, germany, may 19-23, 2019, proceedings, part I, 2019, vol. 11476. https://doi.org/10.1007/978-3-030-17653-2/_5

[8] 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.

[9] S. Research, “The ML-KEM braid protocol,” 2025. https://signal.org/docs/specifications/mlkembraid/

[10] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, “A Formal Security Analysis of the Signal Messaging Protocol.” Cryptology ePrint Archive, Report 2016/1013, 2016. http://eprint.iacr.org/2016/1013

[11] K. Cohn-Gordon, C. Cremers, and L. Garratt, “On Post-Compromise Security.” Cryptology ePrint Archive, Report 2016/221, 2016. http://eprint.iacr.org/2016/221

[12] A. Langley, M. Hamburg, and S. Turner, “Elliptic Curves for Security.” Internet Engineering Task Force; RFC 7748 (Informational); IETF, Jan-2016. http://www.ietf.org/rfc/rfc7748.txt

[13] NIST, “FIPS 180-4. Secure Hash Standard (SHS),” National Institute of Standards & Technology, Gaithersburg, MD, United States, 2012. http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[14] P. Rogaway and T. Shrimpton, “A Provable-security Treatment of the Key-wrap Problem,” in Proceedings of the 24th Annual International Conference on The Theory and Applications of Cryptographic Techniques, 2006. http://web.cs.ucdavis.edu/~rogaway/papers/keywrap.html

[15] NIST, “FIPS 197. Advanced Encryption Standard,” National Institute of Standards & Technology, Gaithersburg, MD, United States, 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[16] B. Kaliski, “PKCS #7: Cryptographic Message Syntax Version 1.5.” Internet Engineering Task Force; RFC 2315 (Informational); IETF, Mar-1998. http://www.ietf.org/rfc/rfc2315.txt

[17] 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

[18] C. Cremers, C. Jacomme, and A. Naska, “Formal analysis of Session-Handling in secure messaging: Lifting security from sessions to conversations,” in 32nd USENIX security symposium (USENIX security 23), 2023. https://www.usenix.org/conference/usenixsecurity23/presentation/cremers-session-handling

[19] N. Borisov, I. Goldberg, and E. Brewer, "Off-the-record Communication, or, Why Not to Use PGP," in Proceedings of the 2004 ACM workshop on privacy in the electronic society, 2004. https://otr.cypherpunks.ca/otr-wpes.pdf

[20] M. Abdalla and M. Bellare, "Increasing the Lifetime of a Key: A Comparative Analysis of the Security of Re-keying Techniques," in Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, 2000. https://cseweb.ucsd.edu/~mihir/papers/rekey.html

[21] B. Olson, "Key Coercion after encrypted message transmission." sci.crypt, 1994. https://groups.google.com/d/topic/sci.crypt/3MJzGwiTZ10/discussion

[22] Wikipedia, "Silent Circle Instant Messaging Protocol — Wikipedia, The Free Encyclopedia." 2016. https://en.wikipedia.org/w/index.php?title=Silent_Circle_Instant_Messaging_Protocol

[23] G. Belvin, "A Secure Text Messaging Protocol." Cryptology ePrint Archive, Report 2014/036, 2014. http://eprint.iacr.org/2014/036

[24] W. M. Petullo, X. Zhang, J. A. Solworth, D. J. Bernstein, and T. Lange, "MinimaLT: Minimal-latency Networking Through Better Security," in Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, 2013. http://doi.acm.org/10.1145/2508859.2516737

[25] A. Langley, "Pond/README.md," 2012. https://github.com/agl/pond/commit/7bb06244b9aa121d367a6d556867992d1481f0c8

[26] M. Rogers and A. Back, "Asynchronous forward secrecy encryption." Cryptography mailing list, 2013. http://lists.randombit.net/pipermail/cryptography/2013-September/005327.html