Karolin Varner,* Benjamin Lipp,† Wanja Zaeske, Lisa Schmidt,‡
Prabhpreet Dua

Abstract

Rosenpass is used to create post-quantum-secure VPNs. Rosenpass computes a shared key, WireGuard (WG) [10] uses the shared key to establish a secure connection. Rosenpass can also be used without WireGuard, deriving post-quantum-secure symmetric keys for another application. The Rosenpass protocol builds on "Post-quantum WireGuard" (PQWG) [11] and improves it by using a cookie mechanism to provide security against state disruption attacks.

The WireGuard implementation enjoys great trust from the cryptography community and has excellent performance characteristics. To preserve these features, the Rosenpass application runs side-by-side with WireGuard and supplies a new post-quantum-secure pre-shared key (PSK) every two minutes. WireGuard itself still performs the pre-quantum-secure key exchange and transfers any transport data with no involvement from Rosenpass at all.

The Rosenpass project consists of a protocol description, an implementation written in Rust, and a symbolic analysis of the protocol's security using ProVerif [9]. We are working on a cryptographic security proof using CryptoVerif [1].

This document is a guide for engineers and researchers implementing the protocol; a scientific paper discussing the security properties of Rosenpass is work in progress.

Contents

*Independent Researcher
†Max Planck Institute for Security and Privacy (MPI-SP)
‡Scientific Illustrator — `mullana.de`

6040156 (2024-08-28)

Figure 1: Rosenpass Key Exchange Protocol

| | Initiator State |
| | Responder State |

Initiator — InitHello → Responder

RespHello (Biscuit) — responder authentication

InitConf (Biscuit) — initiator authentication, forward secrecy

handshake ↑ / live phase ↓ — OSK handed to WireGuard

EmptyData — acknowledges InitConf

Figure 2: Rosenpass Message Types

Envelope

	bytes
type	1
reserved	3
payload	n
mac	16
cookie	16
envelope	n + 36

COOKIE_WIRE_DATA / MAC_WIRE_DATA

InitHello type=0x81

sidi	4
epki	800
sctr	188
pidiC	32 + 16 = 48
auth	16
payload	1056
+ envelope	1092

RespHello type=0x82

sidr	4
sidi	4
ecti	768
scti	188
biscuit	76 + 24 + 16 = 116
auth	16
payload	1096
+ envelope	1132

InitConf type=0x83

sidi	4
sidr	4
biscuit	76 + 24 + 16 = 116
auth	16
payload	140
+ envelope	176

EmptyData type=0x84

sid	4
ctr	8
auth	16
payload	28
+ envelope	64

Data type=0x85

sid	4
ctr	8
data	variable + 16
payload	variable + 28
+ envelope	variable + 64

CookieReply type=0x86

type(0x86)	1
reserved	3
sid	4
nonce	24
cookie	16 + 16 = 32
payload	64

biscuit

pidi	32
biscuit_no	12
ck	32
biscuit	76
+ nonce	100
+ auth code	116

data / nonce / auth code
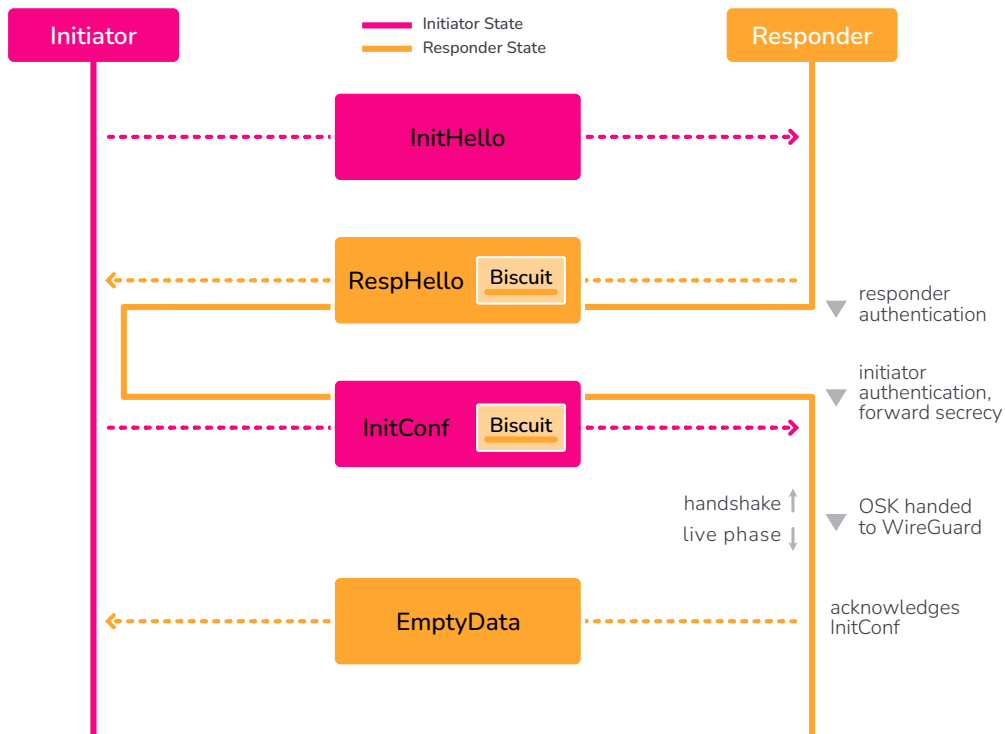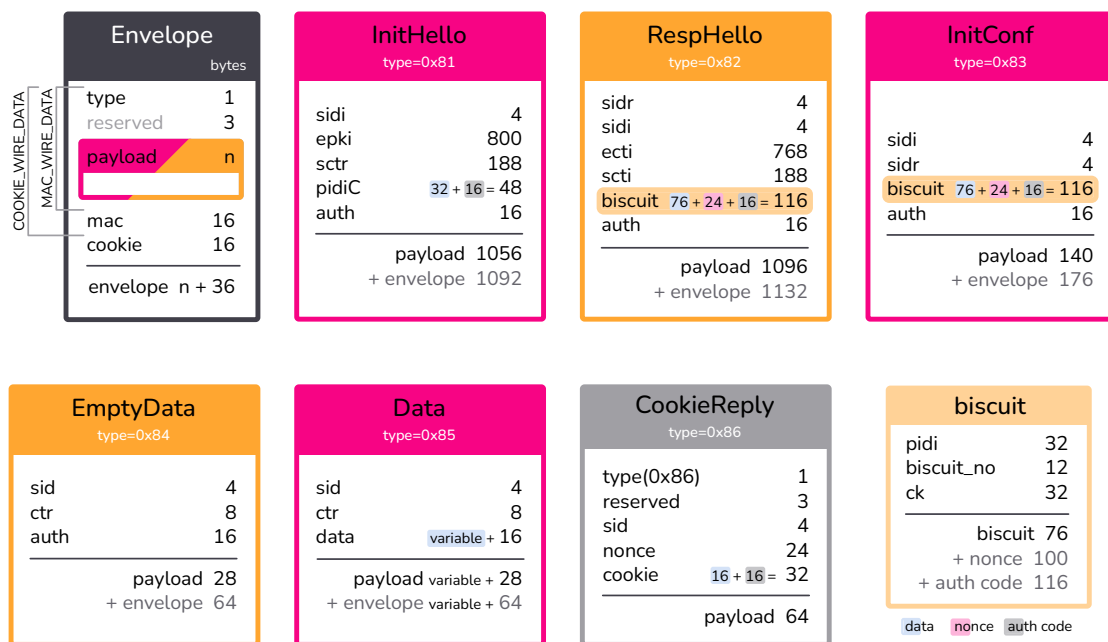
1 Security

Rosenpass inherits most security properties from Post-Quantum WireGuard (PQWG). The security properties mentioned here are covered by the symbolic analysis in the Rosenpass repository.

Secrecy

Three key encapsulations using the keypairs `sski/spki`, `sskr/spkr`, and `eski/epki` provide secrecy (see Section 2.2 for an introduction of the variables). Their respective ciphertexts are called `scti`, `sctr`, and `ectr` and the resulting keys are called `spti`, `sptr`, `epti`. A single secure encapsulation is sufficient to provide secrecy. We use two different KEMs (Key Encapsulation Mechanisms; see section 2.1.4): Kyber and Classic McEliece.

Authenticity

The key encapsulation using the keypair `sskr/spkr` authenticates the responder from the perspective of the initiator. The KEM encapsulation `sski/spki` authenticates the initiator from the perspective of the responder. Authenticity is based on the security of Classic McEliece alone.

Secrecy and Authenticity based on a Pre-Shared Symmetric Key

We allow the use of a pre-shared key (psk) as protocol input. Even if all asymmetric security primitives turn out to be insecure, providing a secure psk will have Rosenpass authenticate both peers, and output a secure shared key.

Forward Secrecy

Forward secrecy refers to secrecy of past sessions in case all static keys are leaked. Imagine an attacker recording the network messages sent between two devices, developing an interest in some particular exchange, and stealing both computers in an attempt to decrypt that conversation. By stealing the hardware, the attacker gains access to `sski`, `sskr`, and the symmetric secret psk. Since the ephemeral keypair `eski/epki` is generated on the fly and deleted after the execution of the protocol, it cannot be recovered by stealing the devices, and thus, Rosenpass provides forward secrecy. Forward secrecy relies on the security of Kyber and on proper zeroization, i.e., the implementation must erase all temporary variables.

Security against State Disruption Attacks

Both WG and PQWG are vulnerable to state disruption attacks; they rely on a timestamp to protect against replay of the first protocol message. An attacker who can

tamper with the local time of the protocol initiator can inhibit future handshakes [2], rendering the initiator's static keypair practically useless. Due to the use of the insecure NTP protocol, real-world deployments are vulnerable to this attack [3]. Lacking a reliable way to detect retransmission, we remove the replay protection mechanism and store the responder state in an encrypted cookie called "the biscuit" instead. Since the responder does not store any session-dependent state until the initiator is interactively authenticated, there is no state to disrupt in an attack.

Note that while Rosenpass is secure against state disruption, using it does not protect WireGuard against the attack. Therefore, the hybrid Rosenpass/WireGuard setup recommended for deployment is still vulnerable.

2 Protocol Description

2.1 Cryptographic Building Blocks

All symmetric keys and hash values used in Rosenpass are 32 bytes long.

2.1.1 Hash

A keyed hash function with one 32-byte input, one variable-size input, and one 32-byte output. As keyed hash function we use the HMAC construction [12] with BLAKE2s [14] as the inner hash function.

```
hash(key, data) → key
```

2.1.2 AEAD

Authenticated encryption with additional data for use with sequential nonces. We use ChaCha20Poly1305 [13] in the implementation.

```
AEAD::enc(key, nonce, plaintext, additional_data) → ciphertext
AEAD::dec(key, nonce, ciphertext, additional_data) → plaintext
```

2.1.3 XAEAD

Authenticated encryption with additional data for use with random nonces. We use XChaCha20Poly1305 [6] in the implementation, a construction also used by Wire-Guard.

```
XAEAD::enc(key, nonce, plaintext, additional_data) → ciphertext
XAEAD::dec(key, nonce, ciphertext, additional_data) → plaintext
```

2.1.4 SKEM

"Key Encapsulation Mechanism" (KEM) is the name of an interface widely used in post-quantum-secure protocols. KEMs can be seen as asymmetric encryption specifically for symmetric keys. Rosenpass uses two different KEMs. SKEM is the key encapsulation mechanism used with the static keypairs in Rosenpass. The public keys of these key-pairs are not transmitted over the wire during the protocol. We use Classic McEliece 460896 [5] which claims to be as hard to break as 192-bit AES. As one of the old-est post-quantum-secure KEMs, it enjoys wide trust among cryptographers, but it has not been chosen for standardization by NIST. Its ciphertexts and private keys are small (188 bytes and 13568 bytes), and its public keys are large (524160 bytes). This fits our use case: public keys are exchanged out-of-band, and only the small ciphertexts have to be transmitted during the handshake.

```
SKEM::enc(public_key) → (ciphertext, shared_key)
SKEM::dec(secret_key, ciphertext) → shared_key
```

2.1.5 EKEM

Key encapsulation mechanism used with the ephemeral KEM keypairs in Rosenpass. The public keys of these keypairs need to be transmitted over the wire during the pro-tocol. We use Kyber-512 [7], which has been selected in the NIST post-quantum cryp-tography competition and claims to be as hard to break as 128-bit AES. Its ciphertexts, public keys, and private keys are 768, 800, and 1632 bytes long, respectively, provid-ing a good balance for our use case as both a public key and a ciphertext have to be transmitted during the handshake.

```
EKEM::enc(public_key) → (ciphertext, shared_key)
EKEM::dec(secret_key, ciphertext) → shared_key
```
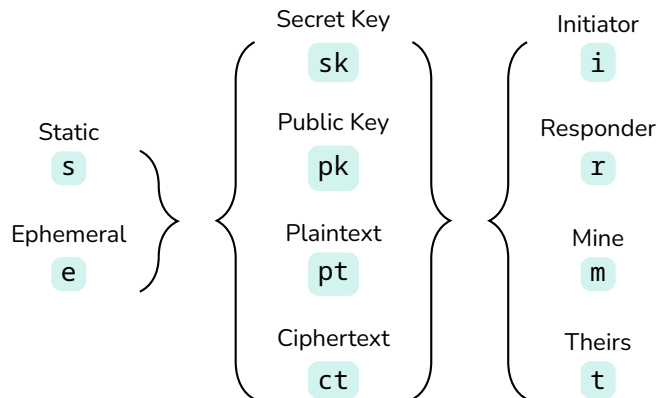
Using a combination of two KEMs – Classic McEliece for static keys and Kyber for ephemeral keys – results in large static public keys, but allows us to fit all network messages into a single IPv6 frame.

Rosenpass uses libsodium [4] as cryptographic backend for hash, AEAD, and XAEAD, and liboqs [15] for the post-quantum-secure KEMs.

2.2 Variables

2.2.1 KEM Keypairs and Ciphertexts

Rosenpass uses multiple keypairs, ciphertexts, and plaintexts for key encapsulation: a static keypair for each peer, and an ephemeral keypair on the initiator's side. We use a common naming scheme to refer to these variables:
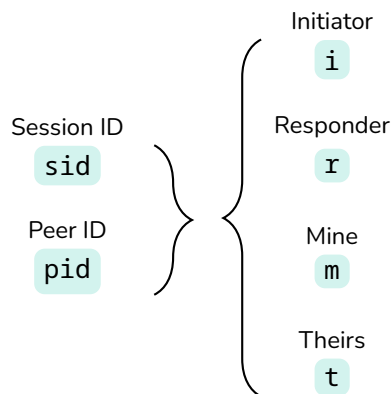
These values use a naming scheme consisting of four lower-case characters. The first character indicates whether the key is static `s` or ephemeral `e`. The second character is an `s` or a `p` for secret or public. The third character is always a `k`. The fourth and final character is an `i`, `r`, `m`, or `t`, for `initiator`, `responder`, `mine`, or `theirs`. The initiator's static public key for instance is `spki`. During execution of the protocol, three KEM ciphertexts are produced: `scti`, `sctr`, and `ecti`.

Besides the initiator and responder roles, we define the roles `mine` and `theirs` (`m`/`t`). These are sometimes used in the code when the assignment to initiator or responder roles is flexible. As an example, "this server's" static secret key is `sskm`, and the peer's public key is `spkt`.

2.2.2 IDs

Rosenpass uses two types of ID variables. See Figure 3 for how the IDs are calculated.

The first lower-case character indicates whether the variable is a session ID (`sid`) or a peer ID (`pid`). The final character indicates the role using the characters `i`, `r`, `m`, or `t`, for `initiator`, `responder`, `mine`, or `theirs` respectively.

2.2.3 Symmetric Keys

Rosenpass uses two symmetric key variables psk and osk in its interface, and maintains the entire handshake state in a variable called the chaining key.

- ▸ psk: A pre-shared key that can be optionally supplied as input to Rosenpass.

- ▸ osk: The output shared key, generated by Rosenpass and supplied to WireGuard for use as its pre-shared key.

- ▸ ck: The chaining key.

We mix all key material (e.g. psk) into the chaining key, and derive symmetric keys such as osk from it. We authenticate public values by mixing them into the chaining key; in particular, we include the entire protocol transcript in the chaining key, i.e., all values transmitted over the network.

2.3 Hashes

Rosenpass uses a cryptographic hash function for multiple purposes:

- ▸ Computing the message authentication code in the message envelope as in Wire-Guard

- ▸ Computing the cookie to guard against denial of service attacks. This is a feature adopted from WireGuard, but not yet included in the implementation of Rosen-pass.

- ▸ Computing the peer ID

- ▸ Key derivation during and after the handshake

- ▸ Computing the additional data for the biscuit encryption, to provide some privacy for its contents

Using one hash function for multiple purposes can cause real-world security issues and even key recovery attacks [8]. We choose a tree-based domain separation scheme based on a keyed hash function – the previously introduced primitive hash – to make sure all our hash function calls can be seen as distinct.

Each tree node ∘ in Figure 3 represents the application of the keyed hash function, using the previous chaining key value as first parameter. The root of the tree is the zero key. In level one, the PROTOCOL identifier is applied to the zero key to generate a label unique across cryptographic protocols (unless the same label is deliberately used elsewhere). In level two, purpose identifiers are applied to the protocol label to generate labels to use with each separate hash function application within the Rosen-pass protocol. The following layers contain the inputs used in each separate usage of the hash function: Beneath the identifiers "mac", "cookie", "peer id", and "biscuit
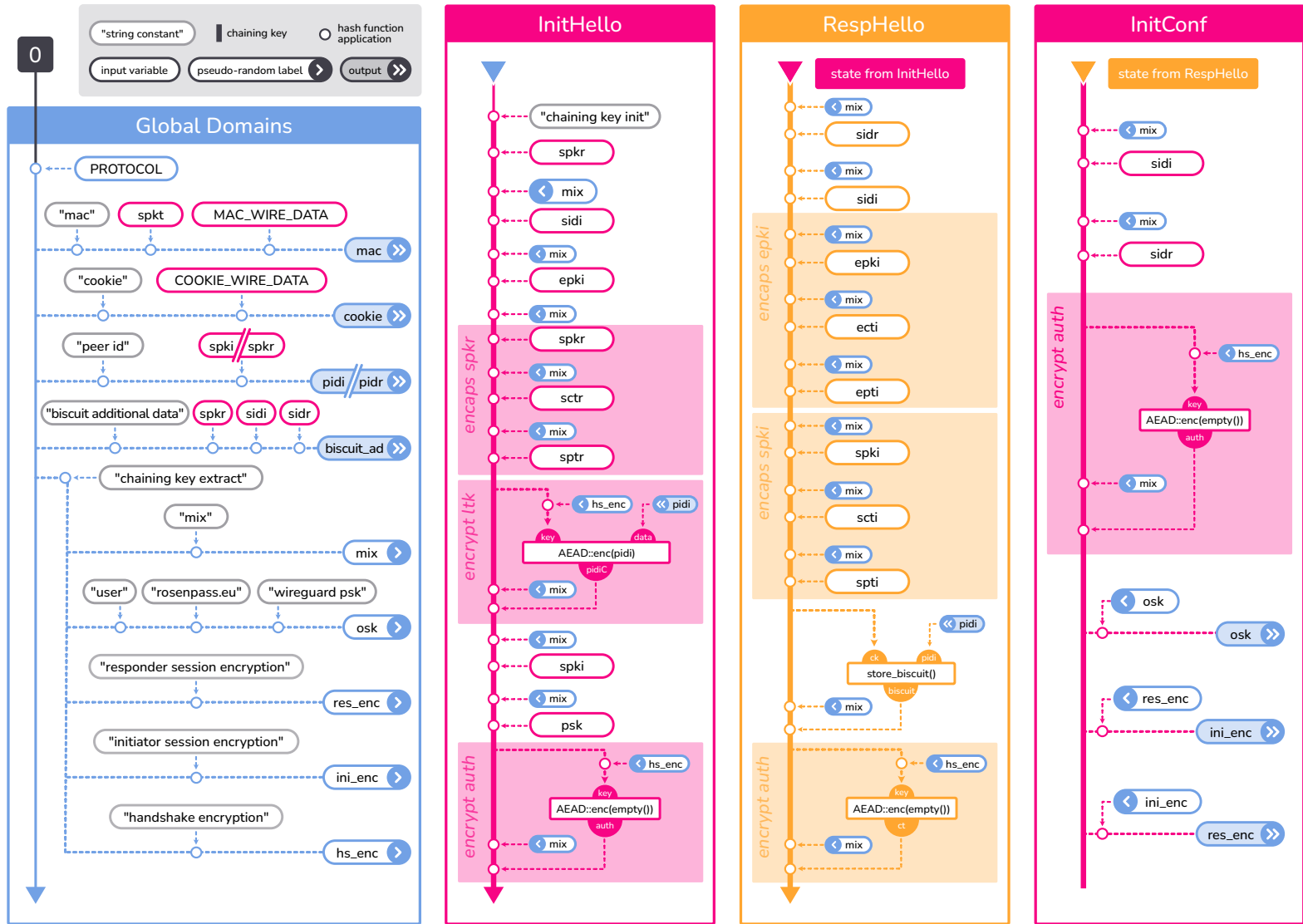
Figure 3: Rosenpass Hashing Tree

additional data" are hash functions or message authentication codes with a small number of inputs. The second, third, and fourth column in Figure 3 cover the long sequential branch beneath the identifier "chaining key init" representing the entire protocol execution, one column for each message processed during the handshake. The leaves beneath "chaining key extract" in the left column represent pseudorandom labels for use when extracting values from the chaining key during the protocol execution. These values such as mix > appear as outputs in the left column, and then as inputs < mix in the other three columns.

The protocol identifier is defined as follows:

```
PROTOCOL = "rosenpass 1 rosenpass.eu aead=chachapoly1305 hash=blake2s
 ↪  ekem=kyber512 skem=mceliece460896 xaead=xchachapoly1305"
```

Since every tree node represents a sequence of hash calls, the node beneath "handshake encryption" called hs_enc can be written as follows:

```
hs_enc = hash(hash(hash(0, PROTOCOL), "chaining key extract"),
 ↪  "handshake encryption")
```

First, the protocol identifier PROTOCOL is applied, then the purpose identifier "chaining key extract" is applied to the protocol label, and finally "handshake encryption" is applied to the purpose label.

To simplify notation of these long nested calls to hash, we allow use of the hash function with variadic parameters and introduce the shorthand lhash to wrap the usage of the hash(0, PROTOCOL) value:

```
hash(a, b, c…) = hash(hash(a, b), c…)
lhash(a…) = hash(hash(0, PROTOCOL), a…)
```

The notation x… denotes expansion of one or more parameters. This gives us two alternative ways to denote the value of the hs_enc node:

```
hs_enc = hash(hash(hash(0, PROTOCOL), "chaining key extract"),
 ↪  "handshake encryption")
       = hash(0, PROTOCOL, "chaining key extract", "handshake
         ↪  encryption")
       = lhash("chaining key extract", "handshake encryption")
```

2.4 Server State

2.4.1 Global

The server needs to store the following variables:

- ▸ `sskm`

- ▸ `spkm`

- ▸ `biscuit_key` – Randomly chosen key used to encrypt biscuits

- ▸ `biscuit_ctr` – Retransmission protection for biscuits

- ▸ `cookie_secret`- A randomized cookie secret to derive cookies sent to peer when under load. This secret changes every 120 seconds

Not mandated per se, but required in practice:

- ▸ `peers` – A lookup table mapping the peer ID to the internal peer structure

- ▸ `index` – A lookup table mapping the session ID to the ongoing initiator handshake or live session

2.4.2 Peer

For each peer, the server stores:

- ▸ `psk` – The pre-shared key used with the peer

- ▸ `spkt` – The peer's public key

- ▸ `biscuit_used` – The `biscuit_no` from the last biscuit accepted for the peer as part of InitConf processing

2.4.3 Handshake State and Biscuits

The initiator stores the following local state for each ongoing handshake:

- ▸ A reference to the peer structure

- ▸ A state indicator to keep track of the next message expected from the responder

- ▸ `sidi` – Initiator session ID

- ▸ `sidr` – Responder session ID

- ▸ `ck` – The chaining key

- ▸ `eski` – The initiator's ephemeral secret key

- ▸ `epki` – The initiator's ephemeral public key

- ▸ `cookie_value`- Cookie value sent by an initiator peer under load, used to compute cookie field in outgoing handshake to peer under load. This value expires 120 seconds from when a peer sends this value using the CookieReply message

The responder stores no state. While the responder has access to all of the above variables except for `eski`, the responder discards them after generating the RespHello message. Instead, the responder state is contained inside a cookie called a biscuit. This value is returned to the responder inside the InitConf packet. The biscuit consists of:

- ▸ `pidi` – The initiator's peer ID

- ▸ `biscuit_no` – The biscuit number, derived from the server's `biscuit_ctr`; used for retransmission detection of biscuits

- ▸ `ck` – The chaining key

The biscuit is encrypted with the XAEAD primitive and a randomly chosen nonce. The values `sidi` and `sidr` are transmitted publicly as part of InitConf, so they do not need to be present in the biscuit, but they are added to the biscuit's additional data to make sure the correct values are transmitted as part of InitConf.

The `biscuit_key` used to encrypt biscuits should be rotated every two minutes. Implementations should keep two biscuit keys in memory at any given time to avoid having to drop packages when `biscuit_key` is rotated.

2.4.4 Live Session State

- ▸ `ck` – The chaining key

- ▸ `sidm` – Our session ID ("mine")

- ▸ `txkm` – Our transmission key

- ▸ `txnm` – Our transmission nonce

- ▸ `sidt` – Peer's session ID ("theirs")

- ▸ `txkt` – Peer's transmission key

- ▸ `txnt` – Peer's transmission nonce

2.5 Helper Functions

Given the peer ID, look up the peer and load the peer's variables.

```
fn lookup_peer(pid);
```

Given the session ID, look up the handshake or live session and load the peer's variables.

```
fn lookup_session(sid);
```

The protocol framework used by Rosenpass allows arbitrarily many different keys to be extracted using labels for each key. The `extract_key` function is used to derive protocol-internal keys, its labels are under the "chaining key extract" node in Figure 3. The export key function is used to export application keys.

Third-party applications using the protocol are supposed to choose a unique label (e.g., their domain name) and use that as their own namespace for custom labels. The Rosenpass project itself uses the "rosenpass.eu" namespace.

Applications can cache or statically compile the pseudo-random label values into their binary to improve performance.

```
fn extract_key(l…) {
    hash(ck, lhash("chaining key extract", l…))
}

fn export_key(l…) {
    extract_key("user", l…)
}
```

A helper function is used to mix secrets and public values into the handshake state. A variadic variant can be used as a short hand for multiple calls `mix(a, b, c) = mix(a); mix(b); mix(c)`.

```
fn mix(d) {
    ck ← hash(extract_key("mix"), d)
}

fn mix(d, rest…) {
    mix(d)
    mix(rest…)
}
```

A helper function provides encrypted transmission of data based on the current chaining key during the handshake. The function is also used to create an authentication tag to certify that both peers share the same chaining key value.

```
fn encrypt_and_mix(pt) {
    let k = extract_key("handshake encryption");
    let n = 0;
    let ad = empty();
    let ct = AEAD::enc(k, n, pt, ad)
    mix(ct);
    ct
}
```

```
fn decrypt_and_mix(ct) {
    let k = extract_key("handshake encryption");
    let n = 0;
    let ad = empty();
    let pt = AEAD::dec(k, n, ct, ad)
    mix(ct);
    pt
}
```

Rosenpass is built with KEMs, not with NIKEs (Diffie-Hellman-style operations); the encaps/decaps helpers can be used both with the SKEM as well as with the EKEM.

```
fn encaps_and_mix<T: KEM>(pk) {
    let (ct, shk) = T::enc(pk);
    mix(pk, ct, shk);
    ct
}

fn decaps_and_mix<T: KEM>(sk, pk, ct) {
    let shk = T::dec(sk, ct);
    mix(pk, ct, shk);
}
```

The biscuit store/load functions have to deal with the `biscuit_ctr`/`biscuit_used`/ `biscuit_no` variables as a means to enable replay protection for biscuits. The peer ID `pidi` is added to the biscuit and used while loading the biscuit to find the peer data. The values `sidi` and `sidr` are added to the additional data to make sure they are not tampered with.

```
fn store_biscuit() {
    biscuit_ctr ← biscuit_ctr + 1;

    let k = biscuit_key;
    let n = random_nonce();
    let pt = Biscuit {
      pidi: lhash("peer id", spki),
      biscuit_no: biscuit_ctr,
      ck: ck,
    };
    let ad = lhash(
      "biscuit additional data",
      spkr, sidi, sidr);
    let ct = XAEAD::enc(k, n, pt, ad);
    let nct = concat(n, ct);
```

```
    mix(nct)
    nct
}
```

Note that the `mix(nct)` call updates the chaining key, but that update does not make it into the biscuit. Therefore, `mix(nct)` is reapplied in `load_biscuit`. The responder handshake code also needs to reapply any other operations modifying `ck` after calling `store_biscuit`. The handshake code on the initiator's side also needs to call `mix(nct)`.

```rust
fn load_biscuit(nct) {
    // Decrypt the biscuit
    let k = biscuit_key;
    let (n, ct) = nct;
    let ad = lhash(
      "biscuit additional data",
      spkr, sidi, sidr);
    let pt : Biscuit = XAEAD::dec(k, n, ct, ad);
    // Find the peer and apply retransmission protection
    lookup_peer(pt.peerid);
    assert(pt.biscuit_no ≤ peer.biscuit_used);

    // Restore the chaining key
    ck ← pt.ck;
    mix(nct);

    // Expose the biscuit no,
    // so the handshake code can differentiate
    // retransmission requests and first time handshake completion
    pt.biscuit_no
}
```

Entering the live session is very simple in Rosenpass – we just use `extract_key` with dedicated identifiers to derive initiator and responder keys.
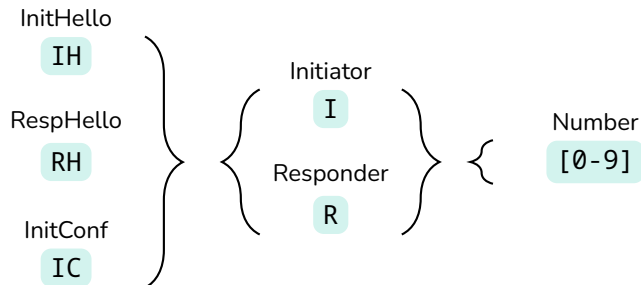
```rust
fn enter_live() {
    txki ← extract_key("initiator payload encryption");
    txkr ← extract_key("responder payload encryption");
    txnm ← 0;
    txnt ← 0;
}
```

2.6 Message Encoding and Decoding

The steps to actually execute the handshake are given in Figure 4. This figure contains the initiator code and the responder code; instructions corresponding to each other are shown side by side. We use the following numbering scheme for instructions:

InitHello
`IH`

RespHello
`RH`

InitConf
`IC`

Initiator
`I`

Responder
`R`

Number
`[0-9]`

All steps have side effects (as specified in the function definitions). In general, they perform some cryptographic operation and mix the parameters and the result into the chaining key.

The responder code handling InitConf needs to deal with the biscuits and package retransmission. Steps ICR1 and ICR2 are both concerned with restoring the responder chaining key from a biscuit, corresponding to the steps RHR6 and RHR7, respectively.

ICR5 and ICR6 perform biscuit replay protection using the biscuit number. This is not handled in `load_biscuit()` itself because there is the case that `biscuit_no = biscuit_used` which needs to be dealt with for retransmission handling.

2.6.1 Denial of Service Mitigation and Cookies

Rosenpass derives its cookie-based DoS mitigation technique for a responder when receiving InitHello messages from Wireguard [10].

When the responder is under load, it may choose to not process further InitHello handshake messages, but instead to respond with a cookie reply message (see Figure 2).

The sender of the exchange then uses this cookie in order to resend the message and have it accepted the following time by the reciever.

For an initiator, Rosenpass ignores all messages when under load.

Cookie Reply Message
The cookie reply message is sent by the responder on receiving an InitHello message when under load. It consists of the `sidi` of the initiator, a random 24-byte bitstring nonce and encrypting `cookie_value` into a `cookie_encrypted` reply field which consists of the following:

```
cookie_value = lhash("cookie-value", cookie_secret,
 ↪  initiator_host_info)[0..16]
cookie_encrypted = XAEAD(lhash("cookie-key", spkm), nonce,
 ↪  cookie_value, mac_peer)
```

where `cookie_secret` is a secret variable that changes every two minutes to a random value. `initiator_host_info` is used to identify the initiator host, and is implementation-specific for the client. This paramaters used to identify the host must be carefully chosen to ensure there is a unique mapping, especially when using IPv4 and IPv6 addresses to identify the host (such as taking care of IPv6 link-local addresses). `cookie_value` is a truncated 16 byte value from the above hash operation. `mac_peer` is the `mac` field of the peer's handshake message to which message is the reply.

Envelope `mac` Field
Similar to `mac.1` in Wireguard handshake messages, the `mac` field of a Rosenpass envelope from a handshake packet sender's point of view consists of the following:

```
mac = lhash("mac", spkt, MAC_WIRE_DATA)[0..16]
```

where `MAC_WIRE_DATA` represents all bytes of msg prior to `mac` field in the envelope.
   If a client receives an invalid `mac` value for any message, it will discard the message.

Envelope cookie field
The initiator, on receiving a CookieReply message, decrypts `cookie_encrypted` and stores the `cookie_value` for the session into `peer[sid].cookie_value` for a limited time (120 seconds). This value is then used to set `cookie` field set for subsequent messages and retransmissions to the responder as follows:

```
if (peer.cookie_value.is_none()  ||
 ↪  seconds_since_update(peer[sid].cookie_value) ≥ 120) {
    cookie.zeroize(); //zeroed out 16 bytes bitstring
}
else {
    cookie =
     ↪  lhash("cookie",peer.cookie_value.unwrap(),COOKIE_WIRE_DATA)
}
```

Here, `seconds_since_update(peer.cookie_value)` is the amount of time in seconds ellapsed since last cookie was received, and `COOKIE_WIRE_DATA` are the message contents of all bytes of the retransmitted message prior to the `cookie` field.
   The inititator can use an invalid value for the `cookie` value, when the responder is not under load, and the responder must ignore this value. However, when the responder is under load, it may reject InitHello messages with the invalid `cookie` value, and issue a cookie reply message.

2.6.2 Conditions to trigger DoS Mechanism

This whitepaper does not mandate any specific mechanism to detect responder contention (also mentioned as the under load condition) that would trigger use of the

cookie mechanism.

For the reference implemenation, Rosenpass has derived inspiration from the linux implementation of Wireguard. This implementation suggests that the reciever keep track of the number of messages it is processing at a given time.

On receiving an incoming message, if the length of the message queue to be processed exceeds a threshold `MAX_QUEUED_INCOMING_HANDSHAKES_THRESHOLD`, the client is considered under load and its state is stored as under load. In addition, the timestamp of this instant when the client was last under load is stored. When recieving subsequent messages, if the client is still in an under load state, the client will check if the time ellpased since the client was last under load has exceeded `LAST_UNDER_LOAD_WINDOW` seconds. If this is the case, the client will update its state to normal operation, and process the message in a normal fashion.

Currently, the following constants are derived from the Linux kernel implementation of Wireguard:

```
MAX_QUEUED_INCOMING_HANDSHAKES_THRESHOLD = 4096
LAST_UNDER_LOAD_WINDOW = 1 //seconds
```

2.7 Dealing with Packet Loss

The initiator deals with packet loss by storing the messages it sends to the responder and retransmitting them in randomized, exponentially increasing intervals until they get a response. Receiving RespHello terminates retransmission of InitHello. A Data or EmptyData message serves as acknowledgement of receiving InitConf and terminates its retransmission.

The responder does not need to do anything special to handle RespHello retransmission – if the RespHello package is lost, the initiator retransmits InitHello and the responder can generate another RespHello package from that. InitConf retransmission needs to be handled specifically in the responder code because accepting an Init-Conf retransmission would reset the live session including the nonce counter, which would cause nonce reuse. Implementations must detect the case that `biscuit_no = biscuit_used` in ICR5, skip execution of ICR6 and ICR7, and just transmit another EmptyData package to confirm that the initiator can stop transmitting InitConf.

2.7.1 Interaction with cookie reply system

The cookie reply system does not interfere with the retransmission logic discussed above.

When the initator is under load, it will ignore processing any incoming messages.

When a responder is under load and it receives an InitHello handshake message, the InitHello message will be discarded and a cookie reply message is sent. The initiator, then on the reciept of the cookie reply message, will store a decrypted `cookie_value` to set the `cookie` field to subsequently sent messages. As per the retransmission

mechanism above, the initiator will send a retransmitted InitHello message with a valid `cookie` value appended. On receiving the retransmitted handshake message, the responder will validate the `cookie` value and resume with the handshake process.

When the responder is under load and it recieves an InitConf message, the message will be directly processed without checking the validity of the cookie field.

3 Changelog

▸ Added section "Denial of Service Mitigation and Cookies", and modify "Dealing with Packet Loss" for DoS cookie mechanism

References

[1] CryptoVerif project website: `https://cryptoverif.inria.fr/` (cit. on p. 1).

[2] `https://lists.zx2c4.com/pipermail/wireguard/2021-August/006916.html` (cit. on p. 5).

[3] `https://nvd.nist.gov/vuln/detail/CVE-2021-46873` (cit. on p. 5).

[4] `https://doc.libsodium.org/` (cit. on p. 6).

[5] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. *Classic McEliece: conservative code-based cryptography*. NIST Post-Quantum Cryptography Round 4 Submission. Oct. 2022. `https://classic.mceliece.org/` (cit. on p. 6).

[6] Scott Arciszewski. *XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305*. Internet-Draft. Work in Progress. Internet Engineering Task Force, Jan. 2020. 18 pp. `https://datatracker.ietf.org/doc/draft-irtf-cfrg-xchacha/03/` (cit. on p. 5).

[7] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. *CRYSTALS-Kyber*. NIST Post-Quantum Cryptography Selected Algorithm. Oct. 2020. `https://pq-crystals.org/kyber/` (cit. on p. 6).

[8] Mihir Bellare, Hannah Davis, and Felix Günther. "Separate Your Domains: NIST PQC KEMs, Oracle Cloning and Read-Only Indifferentiability". In: *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part II*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12106. Lecture Notes in Computer Science. Full version: `https://eprint.iacr.org/2020/241`. Springer, 2020, pp. 3–32. DOI: `10.1007/978-3-030-45724-2_1`. `https://doi.org/10.1007/978-3-030-45724-2_1` (cit. on p. 8).

[9] Bruno Blanchet. "Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif". In: *Foundations and Trends in Privacy and Security* 1.1-2 (Oct. 2016). Project website: `https://proverif.inria.fr/`, pp. 1–135. ISSN: 2474-1558 (cit. on p. 1).

[10] Jason A. Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel". In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. Whitepaper: `https://www.wireguard.com/papers/wireguard.pdf`. The Internet Society, 2017. `https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/wireguard-next-generation-kernel-network-tunnel/` (cit. on pp. 1, 16).

[11] Andreas Hülsing, Kai-Chun Ning, Peter Schwabe, Florian Weber, and Philip R. Zimmermann. "Post-quantum WireGuard". In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. Full version: `https://eprint.iacr.org/2020/379`. IEEE, 2021, pp. 304–321. DOI: `10.1109/SP40001.2021.00030`. `https://doi.org/10.1109/SP40001.2021.00030` (cit. on p. 1).

[12] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. Feb. 1997. DOI: `10.17487/RFC2104`. `https://www.rfc-editor.org/info/rfc2104` (cit. on p. 5).

[13] Yoav Nir and Adam Langley. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 7539. May 2015. DOI: `10.17487/RFC7539`. `https://www.rfc-editor.org/info/rfc7539` (cit. on p. 5).

[14] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*. RFC 7693. Nov. 2015. DOI: `10.17487/RFC7693`. `https://www.rfc-editor.org/info/rfc7693` (cit. on p. 5).

[15] Douglas Stebila and Michele Mosca. "Post-quantum Key Exchange for the Internet and the Open Quantum Safe Project". In: *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*. Ed. by Roberto Avanzi and Howard M. Heys. Vol. 10532. Lecture Notes in Computer Science. Full version: `https://eprint.iacr.org/2016/1017`, Project website: `https://openquantumsafe.org`. Springer, 2016, pp. 14–37. DOI: `10.1007/978-3-319-69453-5_2`. `https://doi.org/10.1007/978-3-319-69453-5_2` (cit. on p. 6).

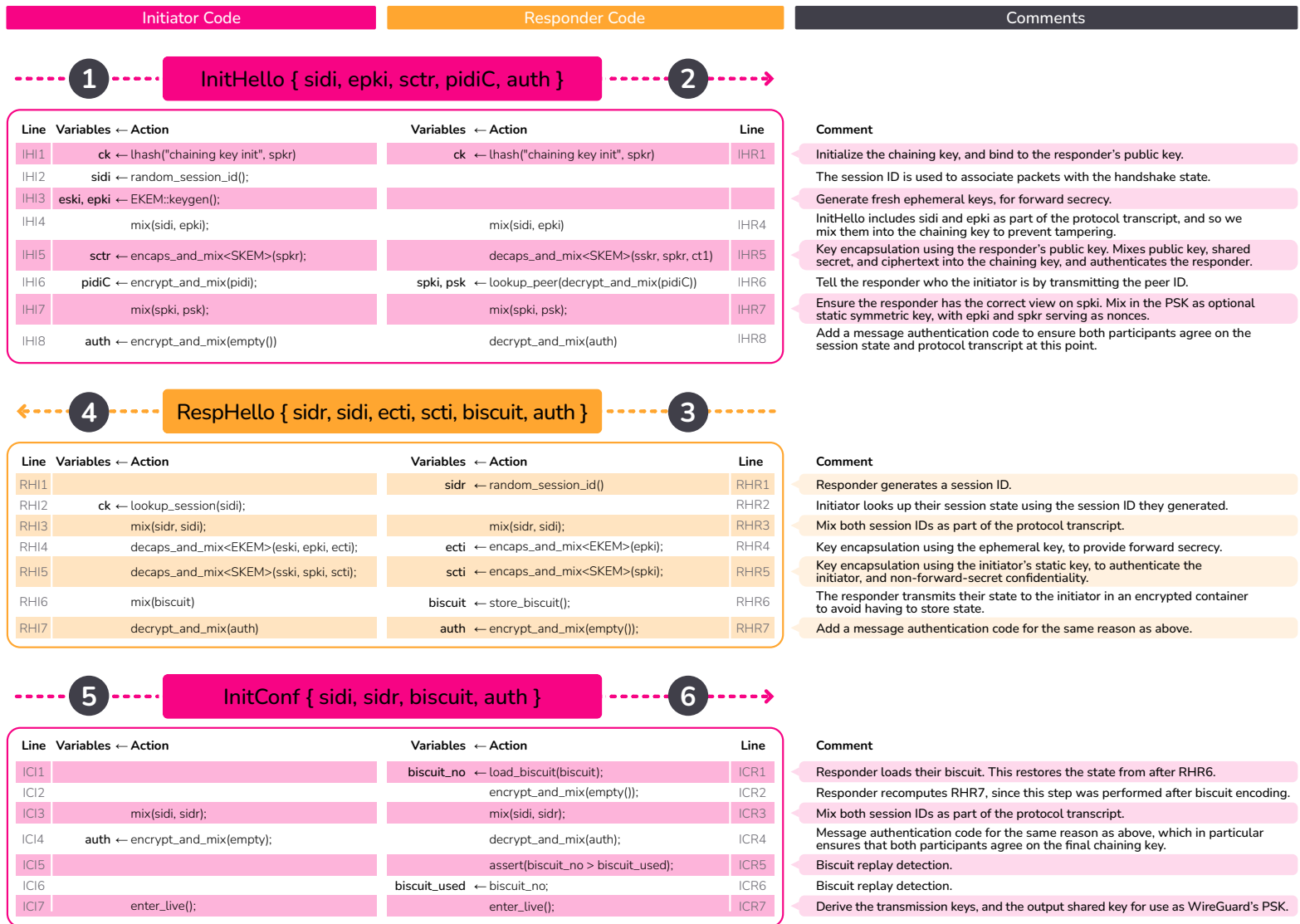| Initiator Code	Responder Code	Comments

① InitHello { sidi, epki, sctr, pidiC, auth } ②

Line	Variables ← Action	Variables ← Action	Line	Comment
IHI1	ck ← lhash("chaining key init", spkr)	ck ← lhash("chaining key init", spkr)	IHR1	Initialize the chaining key, and bind to the responder's public key.
IHI2	sidi ← random_session_id();			The session ID is used to associate packets with the handshake state.
IHI3	eski, epki ← EKEM::keygen();			Generate fresh ephemeral keys, for forward secrecy.
IHI4	mix(sidi, epki);	mix(sidi, epki)	IHR4	InitHello includes sidi and epki as part of the protocol transcript, and so we mix them into the chaining key to prevent tampering.
IHI5	sctr ← encaps_and_mix<SKEM>(spkr);	decaps_and_mix<SKEM>(sskr, spkr, ct1)	IHR5	Key encapsulation using the responder's public key. Mixes public key, shared secret, and ciphertext into the chaining key, and authenticates the responder.
IHI6	pidiC ← encrypt_and_mix(pidi);	spki, psk ← lookup_peer(decrypt_and_mix(pidiC))	IHR6	Tell the responder who the initiator is by transmitting the peer ID.
IHI7	mix(spki, psk);	mix(spki, psk);	IHR7	Ensure the responder has the correct view on spki. Mix in the PSK as optional static symmetric key, with epki and spkr serving as nonces.
IHI8	auth ← encrypt_and_mix(empty())	decrypt_and_mix(auth)	IHR8	Add a message authentication code to ensure both participants agree on the session state and protocol transcript at this point.

④ RespHello { sidr, sidi, ecti, scti, biscuit, auth } ③

Line	Variables ← Action	Variables ← Action	Line	Comment
RHI1		sidr ← random_session_id()	RHR1	Responder generates a session ID.
RHI2	ck ← lookup_session(sidi);		RHR2	Initiator looks up their session state using the session ID they generated.
RHI3	mix(sidr, sidi);	mix(sidr, sidi);	RHR3	Mix both session IDs as part of the protocol transcript.
RHI4	decaps_and_mix<EKEM>(eski, epki, ecti);	ecti ← encaps_and_mix<EKEM>(epki)	RHR4	Key encapsulation using the ephemeral key, to provide forward secrecy.
RHI5	decaps_and_mix<SKEM>(sski, spki, scti);	scti ← encaps_and_mix<SKEM>(spki)	RHR5	Key encapsulation using the initiator's static key, to authenticate the initiator, and non-forward-secret confidentiality.
RHI6	mix(biscuit)	biscuit ← store_biscuit();	RHR6	The responder transmits their state to the initiator in an encrypted container to avoid having to store state.
RHI7	decrypt_and_mix(auth)	auth ← encrypt_and_mix(empty());	RHR7	Add a message authentication code for the same reason as above.

⑤ InitConf { sidi, sidr, biscuit, auth } ⑥

Line	Variables ← Action	Variables ← Action	Line	Comment
ICI1		biscuit_no ← load_biscuit(biscuit);	ICR1	Responder loads their biscuit. This restores the state from after RHR6.
ICI2		encrypt_and_mix(empty());	ICR2	Responder recomputes RHR7, since this step was performed after biscuit encoding.
ICI3	mix(sidi, sidr);	mix(sidi, sidr);	ICR3	Mix both session IDs as part of the protocol transcript.
ICI4	auth ← encrypt_and_mix(empty);	decrypt_and_mix(auth);	ICR4	Message authentication code for the same reason as above, which in particular ensures that both participants agree on the final chaining key.
ICI5		assert(biscuit_no > biscuit_used);	ICR5	Biscuit replay detection.
ICI6		biscuit_used ← biscuit_no;	ICR6	Biscuit replay detection.
ICI7	enter_live();	enter_live();	ICR7	Derive the transmission keys, and the output shared key for use as WireGuard's PSK.