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

Abstract

Rosenpass is a post-quantum-secure authenticated key exchange protocol. Its main practical use case is creating post-quantum-secure VPNs by combining WireGuard and Rosenpass.

In this combination, Rosenpass generates a post-quantum-secure shared key every two minutes that is then used by WireGuard (WG) [12] to establish a secure connection. Rosenpass can also be used without WireGuard, providing post-quantum-secure symmetric keys for other applications, as long as the other application accepts a pre-shared key and provides cryptographic security based on the pre-shared key alone.

The Rosenpass protocol builds on "Post-quantum WireGuard" (PQWG) [13] and improves it by using a cookie mechanism to provide security against state disruption attacks. From a cryptographic perspective, Rosenpass can be thought of as a post-quantum secure variant of the Noise IK[16] key exchange. "Noise IK" means that the protocol makes both parties authenticate themselves, but that the initiator knows before the protocol starts which other party they are communicating with. There is no negotiation step where the responder communicates their identity to the initiator.

The Rosenpass project consists of a protocol description, an implementation written in Rust, and a symbolic analysis of the protocol's security using ProVerif [10]. We are working on a cryptographic security proof using CryptoVerif [1].

This document is a guide for engineers and researchers implementing the protocol.

Contents

[*]Rosenpass e.V., Max Planck Institute for Security and Privacy (MPI-SP)
[†]Rosenpass e.V., Max Planck Institute for Security and Privacy (MPI-SP)
[‡]Scientific Illustrator – `mullana.de`

433ff09 (2025-09-20)

Figure 1: Rosenpass Key Exchange Protocol

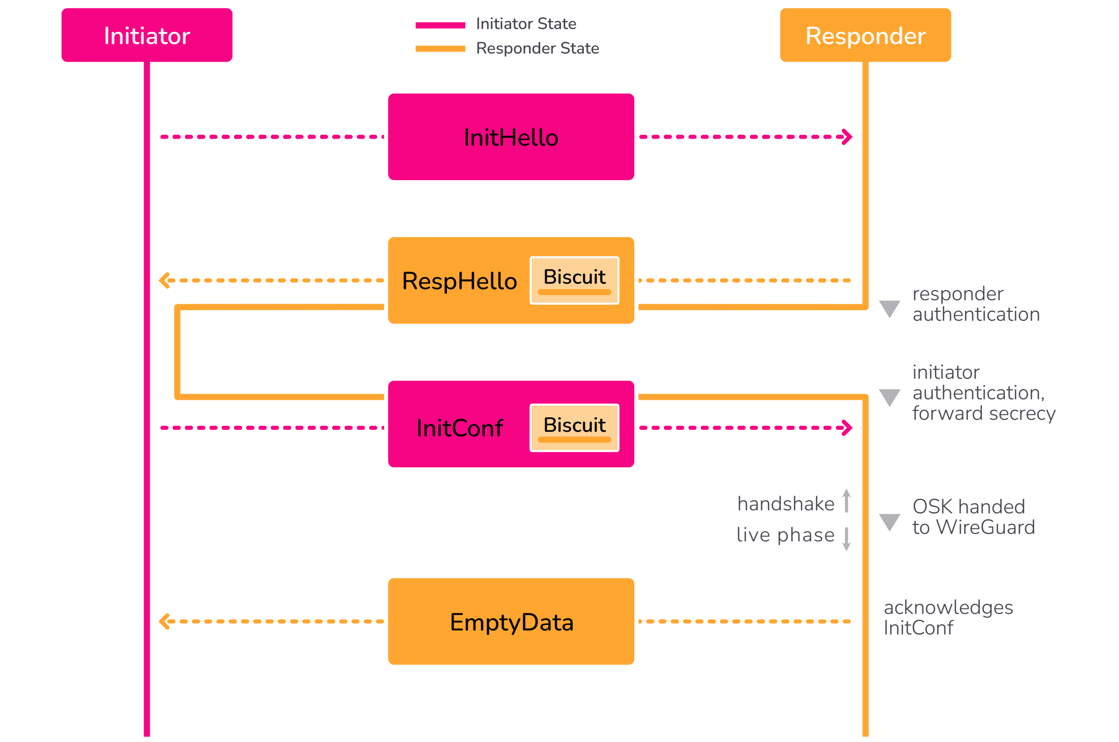
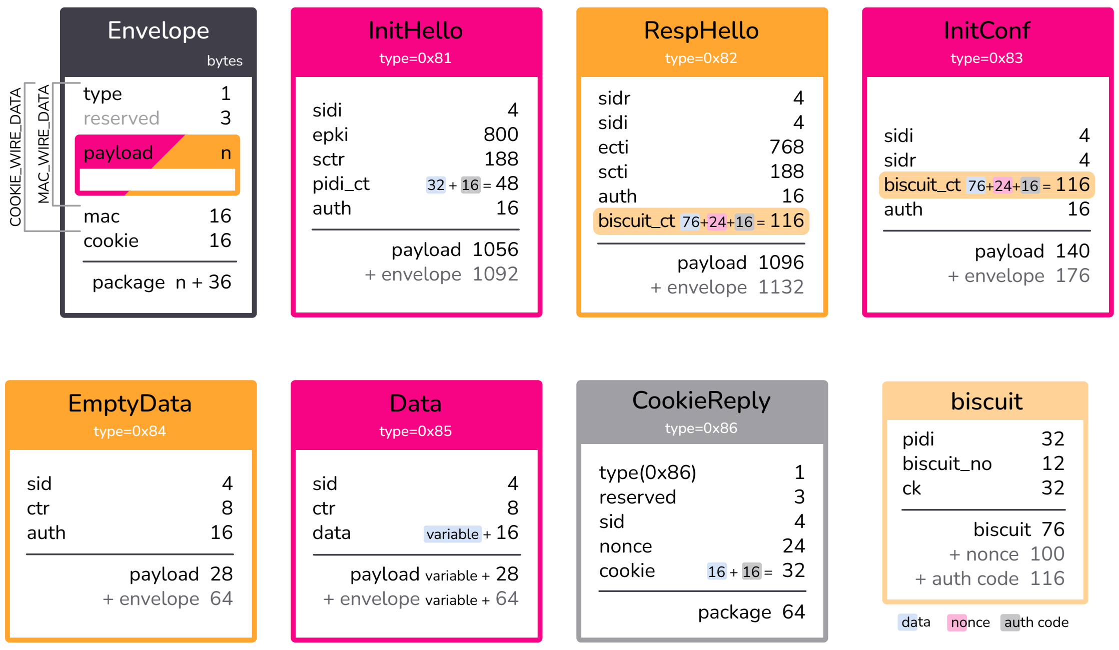


Figure 2: Rosenpass Message Types



Envelope

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

COOKIE_WIRE_DATA, MAC_WIRE_DATA

InitHello type=0x81

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

RespHello type=0x82

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

InitConf type=0x83

sidi	4
sidr	4
biscuit_ct	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
package	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 2.5 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 offer two options that can be configured on a peer-basis, with Blake2b being the default:

1. an **incorrect** HMAC construction [14] with BLAKE2b [17] as the inner hash function. See Sec. 4.1 for details.
2. the SHAKE256 extendable output function (XOF) [18] truncated to a 32-byte output. The result is produced be concatenating the 32-byte input with the variable-size input in this order.

The use of BLAKE2b is being phased out.

```
hash(key, data) → key
```

2.1.2 AEAD

Authenticated encryption with additional data for use with sequential nonces. We use ChaCha20Poly1305 [15] 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 WireGuard.

```
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 keypairs are not transmitted over the wire during the protocol. We use Classic McEliece 460896[1] [5] which claims to be as hard to break as 192-bit AES. As one of the oldest post-quantum-secure KEMs, it enjoys wide trust among cryptographers, but it has not been chosen for standardization by NIST. Its ciphertexts and secret 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 protocol. We use Kyber-512[2] [7], which has been selected in the NIST post-quantum cryptography competition and claims to be as hard to break as 128-bit AES. Its ciphertexts, public keys, and secret keys are 768, 800, and 1632 bytes long, respectively, providing 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
```

[1]The exact Classic McEliece version is from the NIST-Competition, Round 3:
https://classic.mceliece.org/nist/mceliece-20201010.tar.gz

[2]The exact Kyber version is from the NIST-Competition, Round 3:
https://pq-crystals.org/kyber/data/kyber-submission-nist-round3.zip
https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf

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 [19] for the post-quantum-secure KEMs.

2.2 Protocol Roles

The protocol specifies two roles: initiator and responder.

- initiator – The party that starts a handshake.
- responder – The party that does not start a handshake.

All Rosenpass instances operate in either mode; the traditional ”client”/”server” distinction does not apply to the Rosenpass protocol. We sometimes use the term ”server”. In these cases, we generally refer to the ”Rosenpass Server,” as in the application that implements the Rosenpass protocol, not to a server/client distinction.

The initiator is stateful, and directs the handshake process. The responder is stateless for most of the protocol and reacts to the initiator’s messages; this is important to protect our protocol against state disruption (protocol level denial of service) attacks. Since the responder does require some state to complete the protocol, this state is moved into an encrypted cookie, called ”biscuit”.

The number of concurrent responder-role handshakes with another client is unlimited to account for the possibility of an imposter trying to execute a handshake: before completion of said handshake, there is no way to figure out which peer is an imposter and which peer is a legitimate party; any attempt to do so might lead to a state-disruption attack – denial of service on the protocol level.

There is no particular mechanism to negotiate which party acts as initiator and which acts as responder. At startup and when a key exchange is timer-triggered, Rosenpass will *initiate* a key exchange in initiator mode. At startup of another peer, and when they start a timer-triggered key exchange, the local server will *respond* in responder mode.

Implementations must account for one ongoing initiator-role key exchange and many ongoing responder-role key exchanges. Upon receiving a well-formed InitConf package and successfully completing a responder-role key exchange, implementations should abort any ongoing initiator-role key exchange. Implementations should also use different back-off periods depending on whether the handshake was completed in initiator role or in responder role. The following values are used in the Rust reference implementation:

- Initiator rekey interval: 130s
- Responder rekey interval: 120s

In practice these delays cause participants to take turns acting as initiator and acting as responder, since the ten-second difference is usually enough for the handshake with switched roles to complete before the old initiator’s rekey timer goes to zero.

2.3 Packages

The packages, their contents, and their type IDs are graphically represented in Fig. 2. Their purposes are:

- **Envelope** – This is not a package on its own; it is the envelope all the other packages are put into.
- **InitHello** – First package of the handshake, from initiator to responder.
- **RespHello** – Second package of the handshake, from responder to initiator.
- **InitConf** – Third package of the handshake, from initiator to responder.
- **EmptyData** – Empty payload package. Used as acknowledgment to abort data retransmission (see Secs. 2.5.3, 2.11, and function `enter_live()` in Sec. 2.9).
- **Data** – Transmission of actual payload data is not used in Rosenpass, but the package is still specified since it is part of WireGuard (see Sec. 2.5.3 and function `enter_live()` in Sec. 2.9).
- **CookieReply** – Used for proof-of-IP-ownership-based denial-of-service mitigation (see Sec. 2.10.1).
- **biscuit** – This is not a stand-alone package; instead, it is an encrypted fragment present in **RespHello** and **InitConf**.

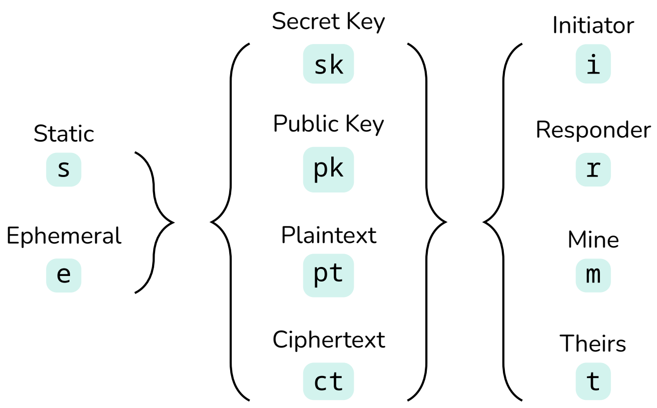
2.4 Endianness

Unless otherwise specified, all integer values in the Rosenpass protocol use little-endian encoding.

2.5 Variables and Domain Separators

2.5.1 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:

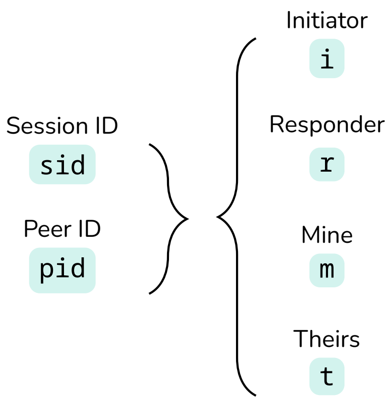


These values use a naming scheme consisting of four lower-case characters. The first character indicates whether the key is static `s` or ephemeral `e`. The second character is an `s` or a `p` for secret or public. The third character is always a `k`. The fourth and final character is an `i`, `r`, `m`, 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, our static secret key is `sskm`, and the peer's public key is `spkt`.

2.5.2 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.

2.5.3 Symmetric Keys

Rosenpass uses two main 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. The main use case is to supply the key to WireGuard for use as its pre-shared key.
- `ck`: The chaining key. This refers to various intermediate keys produced during the execution of the protocol, before the final `osk` is produced.

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.

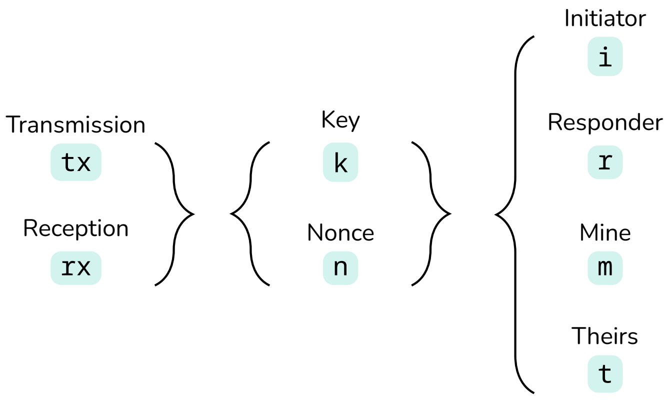
The protocol allows for multiple `osk`s to be generated; each of these keys is labeled with a domain separator to make sure different key usages are always given separate keys. The domain separator for using Rosenpass and WireGuard together is a token generated using the domain separator sequence `["rosenpass.eu", "wireguard psk"]` (see Fig. 3), as described in 3.2. Third-parties using Rosenpass-keys for other purposes are asked to define their own protocol-extensions. Standard protocol extensions are described in 3.

Symmetric Keys and Nonces for payload data transmission
Keys generated by the Rosenpass key exchange could be used for encryption of payload data if post-quantum security but not hybrid post-quantum security is a goal. Despite this, we do not generally offer payload transmission in the protocol. Instead, the Rosenpass protocol focuses on providing a key exchange, letting external applications handle data transmission. When used with WireGuard, the default use case, this integration also ensures hybrid security.

Still we specify the `Data` and `EmptyData` packets. `Data` is not used, but we still specify it as the same packet is also present in WireGuard. `EmptyData` is used for packet retransmission (see Sec. 2.11).

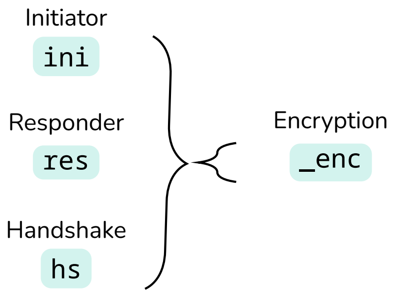
We also specify how symmetric keys are generated for payload encryption. See Sec. #live-session-state and the function `enter_live()` (Sec. 2.9).

Keys and nonces for this purpose use the following naming scheme:



Note that this scheme is deliberately redundant. For instance, when we are the initiator, then `txki = rxki = txkm = rxkt`. I.e. the initiator's transmission key is the responder's reception key. Since we are the initiator, the initiator's transmission key is also the transmission key of `mine` and the reception key of `theirs`.

There also is a – now deprecated – naming scheme:



`ini_enc = txki = rxkr` and `res_enc = txkr = rxki`, but this usage is deprecated. The third name hs_enc is for encryption as part of the key exchange itself; this name is still in use.

2.5.4 Labels

f. 3 specifies multiple domain separators for various uses.

- `PROTOCOL ([0, PROTOCOL])` – The global domain separator; used to generate more domain separators.

Immediately below the global domain separator, you can find:

- `"mac"` – Network package integrity verification and pre-authentication with `spkt`. See Sec. 2.10.1.
- `"cookie"` – Denial of Service mitigation through proof-of-ip ownership. See Sec. 2.10.1.
- `"peer id"` – Generation of peer ids. See Sec. 2.5.2.

- `"biscuit additional data"` – Storing the protocol state in encrypted cookies so the responder is stateless. See Sec. 2.7.3.
- `"chaining key init"` – Starting point for the execution of the actual rosenpass protocol.
- `"chaining key extract"` – Key derivation from the current protocol state, the chaining key. See Sec. 2.5.3.

Below `"chaining key extract"`, there are multiple labels, generating domain separators for deriving keys for various purposes during the execution of the protocol.

It is important to understand that there are two phases for these labels, e.g. applying the `"mix"` label produces a random fixed-size hash value we call `mix`. Not the label `"mix"` but the resulting hash value is used to derive keys during protocol execution. This allows us to use very complicated label structures for key derivation without losing efficiency.

The different labels are:

- `"mix"` – Mixing further values into the chaining key; i.e. into the protocol state.
- `"user"` – Labels for external uses; these are what generate the `osk` (output shared key). See Sec. 2.5.3.
- `"handshake encryption"` – Used when encrypting data using a shared key as part of the protocol execution; e.g. used to generate the `auth` (authentication tag) fields in protocol packages.
- `"initiator handshake encryption"` and `"responder handshake encryption"` – For transmission of data after the key-exchange finishes. See Sec. 2.5.3.

2.6 Hashes

Rosenpass uses a cryptographic hash function for multiple purposes:

- Computing the message authentication code in the message envelope as in WireGuard
- Computing the cookie to guard against denial of service attacks.
- 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

Recall from Section 2.1.1 that rosenpass supports using either BLAKE2b or SHAKE256 as hash function, which can be configured for each peer ID. However, as noted above, rosenpass uses a hash function to compute the peer ID and thus also to access the configuration for a peer ID. This is an issue when receiving an `InitHello`-message, because the correct hash function is not known when a responder receives this message and at the same the responders needs it in order to compute the peer ID and by that also identfy the hash function for that peer. The reference implementation resolves this issue by first trying to derive the peer ID using SHAKE256. If that does not work (i.e. leads to an AEAD decryption error), the reference implementation tries again with BLAKE2b. The reference implementation verifies that the hash function matches the one confgured for the peer. Similarly, if the correct peer ID is not cached when receiving an InitConf message, the reference implementation proceeds in the same manner.

Using one hash function for multiple purposes can cause real-world security issues and even key recovery attacks [9]. 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 $\circ$ 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 Rosenpass 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 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 pseudo-random 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 depends on the hash function used with the respective peer is defined as follows if BLAKE2b [17] is used:

```
PROTOCOL = "Rosenpass v1 mceliece460896 Kyber512 ChaChaPoly1305
 ↪  BLAKE2s"
```

Note that the domain separator used here maintains that BLAKE2s is used, while in reality, we use BLAKE2b. The reason for this is an implementation error. Since fixing this would have led to a breaking change in the Rosenpass reference implementation, and all other known implementations of Rosenpass simply reproduced this error, we chose to harmonize the white paper with the implementation instead of fixing the implementation.

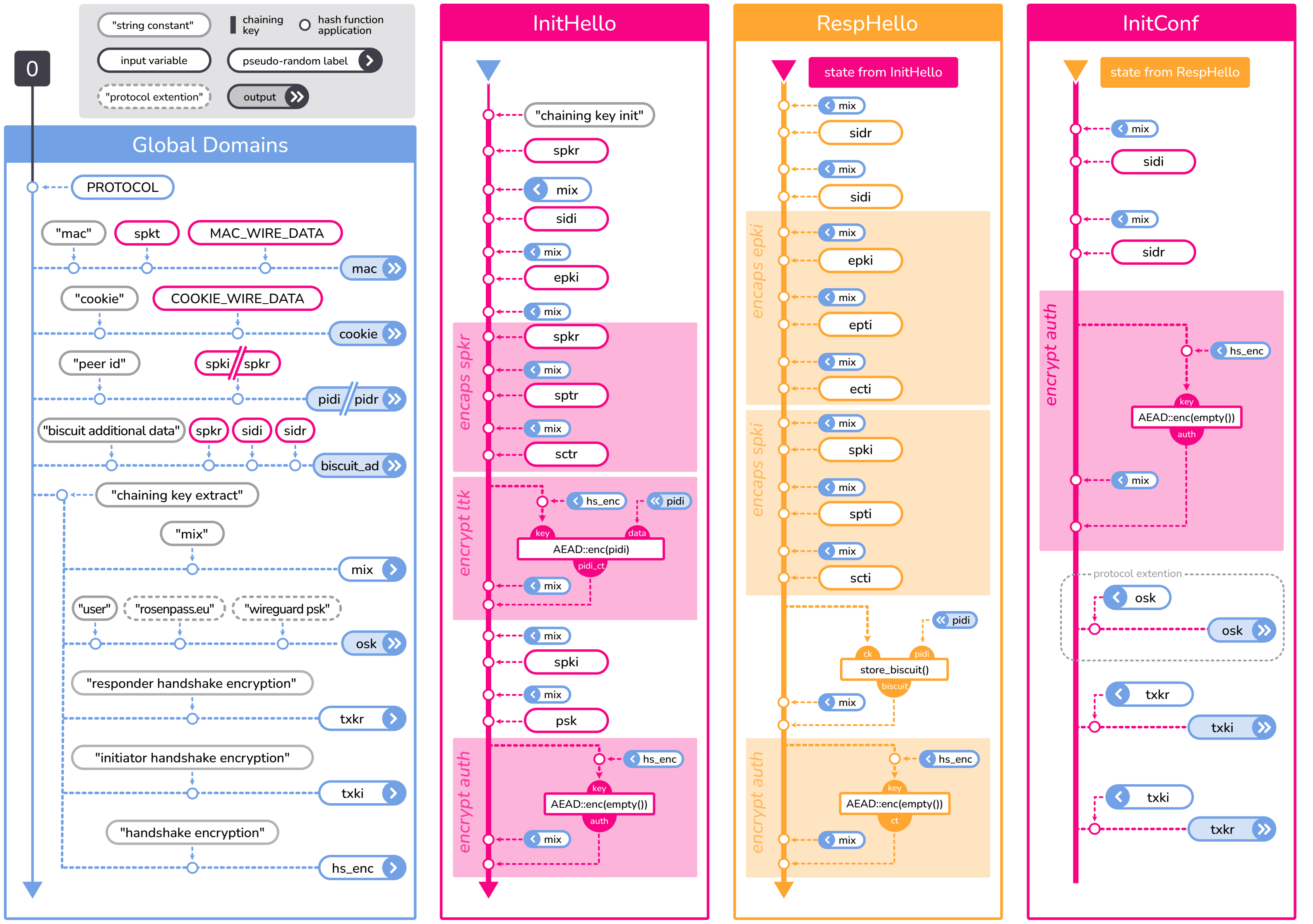


Figure 3: Rosenpass Hashing Tree

If SHAKE256 [18] is used, then BLAKE2s is substituted with SHAKE256:

```
PROTOCOL = "Rosenpass v1 mceliece460896 Kyber512 ChaChaPoly1305
 ↪  SHAKE256"
```

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")
```

2.7 Rosenpass Server State

2.7.1 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

2.7.2 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
- `hash_function` – The hash function, SHAKE256 or BLAKE2b, used with the peer.

2.7.3 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 frequently. Implementations should keep two biscuit keys in memory at any given time to avoid having to drop packages when `biscuit_key` is rotated. The Rosenpass reference implementation retires biscuits after five minutes and erases them after ten.

2.7.4 Live Session State

These variables are used after the handshake terminates for encryption of the **Data** and **EmptyData** packages. **EmptyData** is used as an acknowledgement package to terminate package retransmission (see Sec. 2.11). **Data** would be used for transmission of actual payload, but this feature is currently not specified for Rosenpass. Despite this, we do specify the however as it is also part of WireGuard.

- `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

2.8 Protocol Code

The main reference for how messages are processed in the Rosenpass protocol can be found in Fig. 4. The figure uses Rust-like pseudo code.

2.9 Helper Functions

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

Initiator Code			Responder Code			Comments

1 InitHello { sidi, epki, sctr, pidi_ct, 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, sctr)	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	pidi_ct	← encrypt_and_mix(pidi);	spki, psk	← lookup_peer(decrypt_and_mix(pidi_ct))	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_ct, 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_ct)	biscuit_ct	← 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_ct, auth } 6

Line	Variables	← Action	Variables	← Action	Line	Comment
ICI1			biscuit_no	← load_biscuit(biscuit_ct)	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.

Figure 4: Rosenpass Message Handling Code

```
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 define a protocol extension (see 3) and choose a globally unique label, such as their domain name for custom labels of their own. The Rosenpass project itself uses the `["rosenpass.eu"]` namespace in the WireGuard PSK protocol extension (see 3.2).

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, shk, ct);
    ct
}

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

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 biscuit_ct = concat(n, ct);

    mix(biscuit_ct)
    biscuit_ct
}
```

Note that the `mix(biscuit_ct)` call updates the chaining key, but that update does not make it into the biscuit. Therefore, `mix(biscuit_ct)` 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(biscuit_ct)`.

```
fn load_biscuit(biscuit_ct) {
    // Decrypt the biscuit
    let k = biscuit_key;
    let concat(n, ct) = biscuit_ct;
    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.pidi);

    // In December 2024, the InitConf retransmission mechanism was
     ↪  redesigned
    // in a backwards-compatible way. See the changelog.
    //
    // -- 2024-11-30, Karolin Varner
    if (protocol_version!(< "0.3.0")) {
        // Ensure that the biscuit is used only once
        assert(pt.biscuit_no ≥ peer.biscuit_used);
    }

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

    // 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.

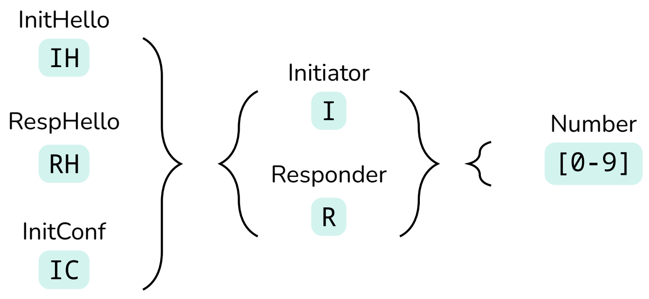
```
fn enter_live() {
    txki ← extract_key("initiator payload encryption");
    txkr ← extract_key("responder payload encryption");
    txnm ← 0;
    txnt ← 0;

    // Setup output keys for protocol extensions such as the
    // WireGuard PSK protocol extension.
    setup_osks();
}
```

The final step `setup_osks()` can be defined by protocol extensions (see 3) to set up osks for custom use cases. By default, the WireGuard PSK (see 3.2) is active.

```
fn setup_osks() {
    ... // Defined by protocol extensions
}
```

2.10 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:



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.

2.10.1 Denial of Service Mitigation and Cookies

Rosenpass derives its cookie-based DoS mitigation technique for a responder when receiving InitHello messages from WireGuard [12].

This is currently implemented in the Rosenpass implementation but still considered an experimental feature and not enabled by default.

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 receiver.

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. Moreover, `lhash` is always instantiated with SHAKE256 when computing `cookie_value` for compatability reasons. `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 elapsed 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.

2.10.2 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 receiver 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 elapsed 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
```

2.11 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 uses less complex form of the same mechanism: The responder never retransmits RespHello, instead the responder generates a new RespHello message if InitHello is retransmitted. Responder confirmation messages of completed handshake (EmptyData) messages are retransmitted by storing the most recent InitConf messages (or their hashes) and caching the associated EmptyData messages. Through this cache, InitConf retransmission is detected and the associated EmptyData message is retransmitted.

2.11.1 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.

2.12 Timers

The Rosenpass protocol uses various timer-triggered events during its operation. This section provides a listing of the timers used and gives the values used in the reference implementation. Other implementations may choose different values.

2.12.1 Rekeying

Period after which the previous responder starts a new handshake in initiator role; period after which the previous initiator starts a new handshake in initiator role again; period after which a peer rejects an existing shared key.

```
REKEY_AFTER_TIME_RESPONDER = 120s
REKEY_AFTER_TIME_INITIATOR = 130s
REJECT_AFTER_TIME = 180s
```

2.12.2 Biscuits

Period after which the biscuit key is rotated.

```
BISCUIT_EPOCH = 300s
```

2.12.3 Retransmission

Delay after which all retransmission attempts are aborted; exponential backoff factor for retransmission delay; initial (minimum) retransmission delay; final (maximum) retransmission delay; retransmission jitter/variance factor.

```
RETRANSMIT_ABORT        = 120s
RETRANSMIT_DELAY_GROWTH = 2
RETRANSMIT_DELAY_BEGIN  = 500ms
RETRANSMIT_DELAY_END    = 10s
RETRANSMIT_DELAY_JITTER = 0.5
```

3 Protocol extensions

The main extension point for the Rosenpass protocol is to generate osks (speak output shared keys, see Sec. 2.5.3) for purposes other than using them to secure WireGuard. By default, the Rosenpass application generates keys for the WireGuard PSK (see 3.2). It would not be impossible to use the keys generated for WireGuard in other use cases, but this might lead to attacks[9]. Specifying a custom protocol extension in practice just means settling on alternative domain separators (see Sec. 2.5.3, Fig. 3).

3.1 Using custom domain separators in the Rosenpass application

The Rosenpass application supports protocol extensions to change the OSK domain separator without modification of the source code.

The following example configuration file can be used to execute Rosenpass in outfile mode with custom domain separators. In this mode, the Rosenpass application will write keys to the file specified with key_out and send notifications when new keys are exchanged via standard out. This can be used to embed Rosenpass into third-party application.

```
# peer-a.toml
public_key = "peer-a.pk"
secret_key = "peer-a.sk"
listen = ["[::1]:6789"]
verbosity = "Verbose"

[[peers]]
public_key = "peer-b.pk"
key_out = "peer-a.osk" # path to store the key
osk_organization = "myorg.com"
osk_label = ["My Custom Messenger app", "Backend VPN Example
 ↪  Subusecase"]
```

3.2 Extension: WireGuard PSK

The WireGuard PSK protocol extension is active by default; this is the mode where Rosenpass is used to provide post-quantum security for WireGuard. Hybrid security (i.e. redundant pre-quantum and post-quantum security) is achieved because WireGuard provides pre-quantum security, with or without Rosenpass.

This extension uses the `"rosenpass.eu"` namespace for user-labels and specifies a single additional user-label:

- `["rosenpass.eu", "wireguard psk"]`

The label's full domain separator is

- `[PROTOCOL, "user", "rosenpass.eu", "wireguard psk"]`

and can be seen in Figure 3.

We require two extra per-peer configuration variables:

- `wireguard_interface` — Name of a local network interface. Identifies local WireGuard interface we are supplying a PSK to.
- `wireguard_peer` — A WireGuard public key. Identifies the particular WireGuard peer whose connection we are supplying PSKs for.

When creating the WireGuard interface for use with Rosenpass, the PSK used by Wire-Guard must be initialized to a random value; otherwise, WireGuard can establish an insecure key before Rosenpass had a change to exchange its own key.

```
fn on_wireguard_setup() {
    // We use a random PSK to make sure the other side will never
    // have a matching PSK when the WireGuard interface is created.
    //
    // Never use a fixed value here as this would lead to an attack!
    let fake_wireguard_psk = random_key();

    // How the interface is create
    let wg_peer = WireGuard::setup_peer()
        .public_key(wireguard_peer)
        ... // Supply any custom peerconfiguration
        .psk(fake_wireguard_psk);

    // The random PSK must be supplied before the
    // WireGuard interface comes up
    WireGuard::setup_interface()
        .name(wireguard_interface)
        ... // Supply any custom configuration
        .add_peer(wg_peer)
        .create();
}
```

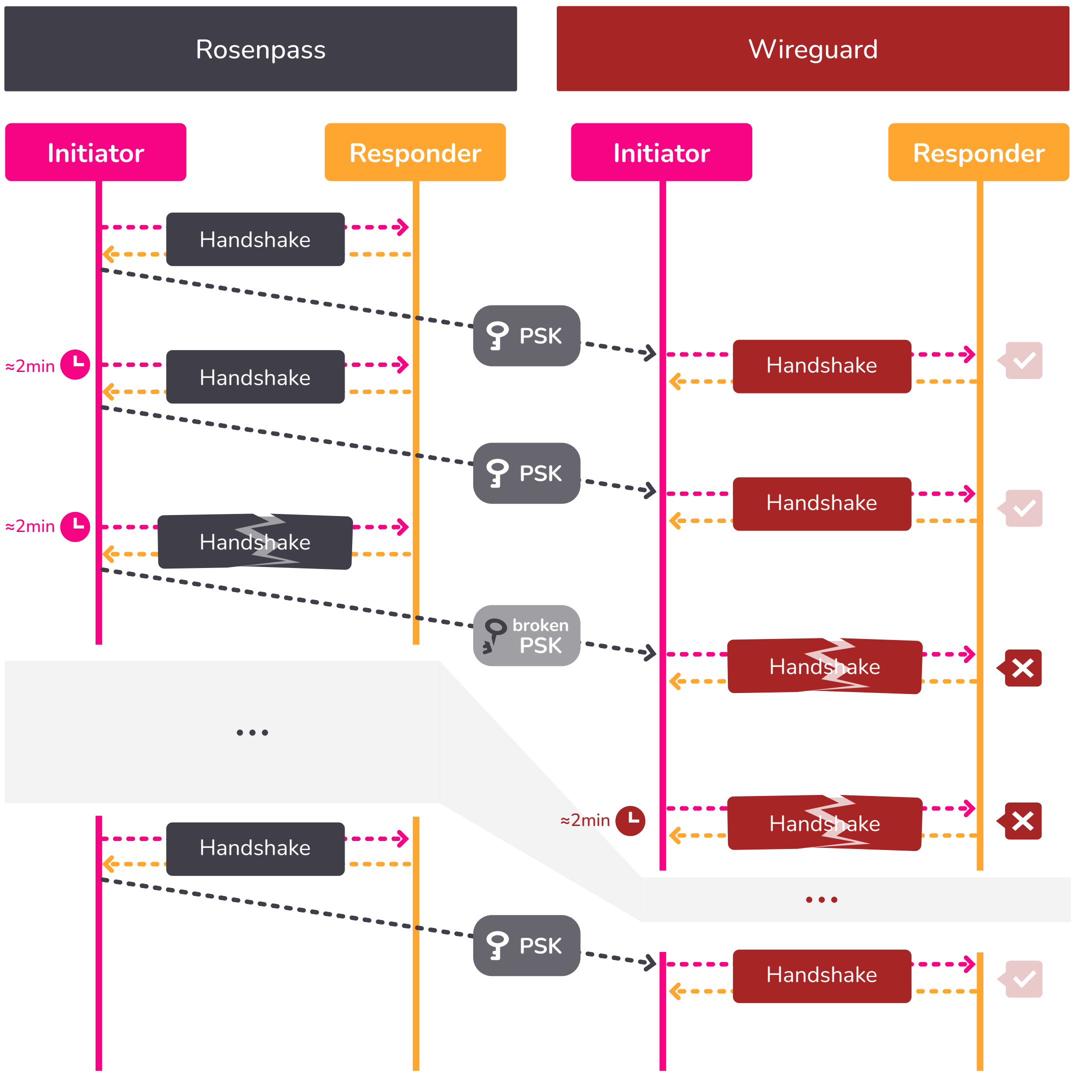
Every time a key is successfully negotiated, we upload the key to WireGuard. For this protocol extension, the `setup_osks()` function is thus defined as:

```
fn setup_osks() {
    // Generate WireGuard OSK (output shared key) from Rosenpass'
    // perspective, respectively the PSK (preshared key) from
    // WireGuard's perspective
    let wireguard_psk = export_key("rosenpass.eu", "wireguard psk");

    /// Supply the PSK to WireGuard
    WireGuard::get_interface(wireguard_interface)
        .get_peer(wireguard_peer)
        .set_psk(wireguard_psk);
}
```

The Rosenpass protocol uses key renegotiation, just like WireGuard. If no new osk is produced within a set amount of time, the OSK generated by Rosenpass times out. In this case, the WireGuard PSK must be overwritten with a random key. This interaction is visualized in Figure 5.

```
fn on_key_timeout() {
    // Generate a random - deliberately invalid - WireGuard PSK.
    // Never use a fixed value here as this would lead to an attack!
    let fake_wireguard_psk = random_key();

    // Securely erase the PSK currently used by WireGuard by
    // overwriting it with the fake key we just generated.
    WireGuard::get_interface(wireguard_interface)
        .get_peer(wireguard_peer)
        .set_psk(fake_wireguard_psk);
}
```

Figure 5: Rosenpass + WireGuard: Hybrid Security



4 Errata

4.1 Incorrect HMAC, Hash Function Choice

Initially, we chose to use `HMAC+BLAKE2s` for our message authentication code, mostly as a form of cargo cult. WireGuard used BLAKE2s, so we should use it too. BLAKE2 supports a directly keyed mode, so there is not much reason to prefer rolling your own using HMAC from a security standpoint.

It seems likely that WireGuard used HMAC as a heuristic security measure. Message authentication codes, keyed hash functions, had long been constructed by combining HMAC with a hash function; why change that? And there actually is a good reason to use HMAC: Merkle-Damgard constructions have long been the norm for hash functions; their usage was even standardized as MD5 or SHA-2. But Merkle-Damgard constructions are susceptible to extension attacks, where you can calculate `H(message || suffix)` assuming `H(message)` is known to you. HMAC fixes this issue[11, 8].

But SHA-3 (or SHAKE) and BLAKE2 depart from this long-standing status quo: these hash functions are not based on Merkle-Damgard and they are deliberately designed so they are not susceptible to length extension attacks. On top of this, both schemes provide a keyed mode as a feature of the hash function. At this point it makes much more sense to require a keyed hash function, satisfying the PRF ("pseudo random function") security property and the PRF-SWAP security property[13] instead of building our own keyed hash from a hash function. HMAC can still be used; if someone wanted to operate Rosenpass with SHA2, the best way to do it would be using `HMAC-SHA512` as the underlying keyed hash. We just also allow using `SHAKE256` without an extra application of HMAC.

Unfortunately, there were a couple of errors in the implementation: we should have used BLAKE2s like WireGuard; instead, we used BLAKE2b. We should have implemented HMAC properly, but we failed to do so. For a fixed-length, 32 byte key and a 32 byte block size, the HMAC function is specified as:

```
type Key = [u8; 32];
type HashFunction = Fn(&[u8]) → Key;

const INNER_PAD: [u8; KEY_LEN] = [0x36u8; KEY_LEN];
const OUTER_PAD: [u8; KEY_LEN] = [0x5Cu8; KEY_LEN];

fn hmac<Hash: HashFunction>(h: Hash, key: Key, data: &[u8]) → Key {
    // `^` denotes XOR, `||` denotes concatenation

    let inner_key = key ^ INNER_PAD;
    let outer_key = key ^ OUTER_PAD;

    let inner_hash = h(inner_key || data);
    let outer_hash = h(outer_key || inner_hash);

    return outer_hash;
}
```

Instead of implementing this function, we somehow lost track of the fact that HMAC uses concatenation to combine the keys with its data, and instead we built a construction around BLAKE2b in keyed hash mode. That is, we replaced the concatenation with calls to the keyed version of our hash:

```
type Key = [u8; 32];
type KeyedHashFunction = Fn(Key, &[u8]) → Key;
```

```
const INNER_PAD: [u8; KEY_LEN] = [0x36u8; KEY_LEN];
const OUTER_PAD: [u8; KEY_LEN] = [0x5Cu8; KEY_LEN];

fn incorrect_rosenpass_hmac<KeyedHash: KeyedHashFunction>(kh:
 ↪  KeyedHashFunction, key: Key, data: &[u8]) → Key {
    // `^` denotes XOR, `||` denotes concatenation

    let inner_key = key ^ INNER_PAD;
    let outer_key = key ^ OUTER_PAD;

    let inner_hash = kh(inner_key, data);
    let outer_hash = kh(outer_key, inner_hash);

    return outer_hash;
}
```

We therefore add this section explaining our incorrect HMAC usage to harmonize the white paper with the implementation. To ensure compatibility with the existing versions of Rosenpass, you have to replicate this incorrect variant of HMAC.

Neither mistake is assumed to cause security issues. BLAKE2b is a secure hash function. There is no reason to assume that our incorrect variant of HMAC-BLAKE2b would be insecure; it is, however, non-standard and needlessly complicates the protocol. We are therefore phasing out usage of HMAC-BLAKE2b in favor of us using SHAKE256 as our keyed hash of choice.

5 Changelog

5.0.1 0.3.x

2025-08-10 – Applying fixes from Steffen Vogel proof reading of the whitepaper

Author: Karolin varner
Issue: #68[3]
PR: #664[4]

Early in the project lifetime, Steffen Vogel successfully implemented a [port of the Rosenpass protocol in go[5]. This implementation has not received an in-depth review from a cryptography implementation perspective, which is why we (the Rosenpass project) are not yet recommending this implementation for production usage; still, creating this implementation was a great achievement.

[3] https://github.com/rosenpass/rosenpass/issues/68
[4] https://github.com/rosenpass/rosenpass/
[5] https://github.com/cunicu/go-rosenpass

During the process, Steffen discovered a large number of possible improvements for the whitepaper. With this update, we are addressing those issues.

This process also ensures that the world knows, that I have ADHD and makes me fix all the little mistakes I could not spot even on the seventh review of the whitepaper.

Changes, in particular:

1. Added a comprehensive reference about labels used in the protocol
2. Added a comprehensive reference about symmetric keys and nonces used for encryption/decryption (`txki`, `txni`, `ini_enc`, `hs_enc`, ...)
3. Added a comprehensive reference about packages used.
4. Added an explaining paragraph to section ”Live Session State”.
5. Added a section about protocol roles.
6. Brief section about endianness.
7. In Fig. 5: Rosenpass Message Handling Code; in IHR5 we replace

    ```
        decaps_and_mix<SKEM>(sskr, spkr, ct1)
    ```

    by

    ```
        decaps_and_mix<SKEM>(sskr, spkr, sctr)
    ```

8. In `load_biscuit()`, there was a typo doing an incorrect comparison between `biscuit_no` and `biscuit_used`. This is not a security issue, as a verbatim implementation would simply have lead to a non-functional implementation. We replace

    ```
        assert(pt.biscuit_no ≤ peer.biscuit_used);
    ```

    by

    ```
        assert(pt.biscuit_no ≥ peer.biscuit_used);
    ```

9. In the whitepaper we used the labels `"initiator session encryption"` and `"responder session encryption"`, but in the implementation we used `"initiator handshake encryption"` and `"responder handshake encryption"`. While the whitepaper was correct and the implementation was not, we opt to harmonize the whitepaper with the implementation to avoid a breaking change.

10. The protocol strings used in the whitepaper where different to the ones used in the implementation. We harmonize the two by updating the whitepaper to reflect the protocol identifier used in the implementation. We substitute

    > The protocol identifier depends on the hash function used with the respective peer is defined as follows if BLAKE2s is used:
    >
    > ```
    > PROTOCOL = "rosenpass 1 rosenpass.eu
    >  ↪ aead=chachapoly1305 hash=blake2s ekem=kyber512
    >  ↪ skem=mceliece460896 xaead=xchachapoly1305"
    > ```
    >
    > If SHAKE256 is used, `blake2s` is replaced by `shake256` in `PROTOCOL`.

    with

    > The protocol identifier depends on the hash function used with the respective peer is defined as follows if BLAKE2s is used:
    >
    > ```
    > PROTOCOL = "Rosenpass v1 mceliece460896 Kyber512
    >  ↪ ChaChaPoly1305 BLAKE2s"
    > ```
    >
    > If SHAKE256 is used, then `BLAKE2s` is substituted with `SHAKE256`:
    >
    > ```
    > PROTOCOL = "Rosenpass v1 mceliece460896 Kyber512
    >  ↪ ChaChaPoly1305 SHAKE256"
    > ```

11. The whitepaper stated that Rosenpass uses BLAKE2s, while the implementation used BLAKE2b; we update the whitepaper to reflect that reality. The places where this substitution happened are a bit too numerous to count them all here. On top of this, we added the following paragraph to explain the discrepancy between `PROTOCOL` and actual hash function used:

    > Note that the domain separator used here maintains that BLAKE2s is used, while in reality, we use BLAKE2b. The reason for this is an implementation error. Since fixing this would have led to a breaking change in the Rosenpass reference implementation, and all other known implementations of Rosenpass simply reproduced this error, we chose to harmonize the white paper with the implementation instead of fixing the implementation.

12. Added a section to explain and specify our incorrect implementation of HMAC-BLAKE2b.

13. In `encaps_and_mix()`/`decaps_and_mix()` the whitepaper stated that public key, ciphertext, and shared key are mixed into the chaining key in that order, but the implementation used a different order: public key, shared key, and ciphertext (shared

key and ciphertext are swapped). We harmonize the white paper with the implementation.

14. In the white paper, in package `RespHello` the field `auth` was indicated to come after `biscuit`, but in the implementation, `auth` came first and `biscuit` was last. The semantics of how fields in Rosenpass messages are processed generally demand that fields are processed in the order they appear in the message, so having `biscuit` first and `auth` second—as was done in the white paper—would be correct; still, we harmonize the white paper with the implementation.

15. Fix a discrepancy with regard to biscuit key life times.

    > 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.

    by

    > The `biscuit_key` used to encrypt biscuits should be rotated frequently. Implementations should keep two biscuit keys in memory at any given time to avoid having to drop packages when `biscuit_key` is rotated. The Rosenpass reference implementation retires biscuits after five minutes and erases them after ten.

16. Point out explicitly that we use KEMs from NIST-Competition Round 3. Include links to the competition submission packages. Update citations to reflect the exact specification version.

17. Consistent naming convention. Always use the term `secret` key, never `private` key.

18. `pidiC` -> `pidi_ct`; to make it clearer that this is a cipher text

19. Where we refer to the biscuit ciphertext, we now use the term `biscuit_ct`. Previously we had used various variable names such as `nct` (nonce followed by cipher text) or just plain `biscuit`.

20. In `load_biscuit`, we make it clear that destructuring of `biscuit_ct` destructures a concatenation.

    ```
    let (n, ct) = biscuit_ct;
    ```

    with

    ```
    let concat(n, ct) = biscuit_ct;
    ```

21. Added a section about timers used in the Rosenpass protocol

Additional changes (also motivated by a close review, but not reported by Steffen):

1. f. 2 ”Rosenpass Message Types”, CookieReply package. Renamed the length sum from payload to package.
2. f. 2 ”Rosenpass Message Types”, Envelope package. Renamed the length sum from envelope to package.
3. In `load_biscuit()` fix a naming typo:

   ```
   lookup_peer(pt.peerid);
   ```

   with

   ```
   lookup_peer(pt.pidi);
   ```

4. Remove reference to the proof-of-IP-ownership-based DoS mitigation feature not being implemented. Add a notice, that the feature is currently experimental.
5. Fixed a few typos and capitalization issues

2025-06-24 – Specifying the osk used for WireGuard as a protocol extension

Author: Karolin varner
PR: #664[6]

We introduce the concept of protocol extensions to make the option of using Rosenpass for purposes other than encrypting WireGuard more explicit. This captures the status-quo in a better way and does not constitute a functional change of the protocol.

When we designed the Rosenpass protocol, we built it with support for alternative osk-labels in mind. This is why we specified the domain separator for the osk to be `[PROTOCOL, "user", "rosenpass.eu", "wireguard psk"]`. By choosing alternative values for the namespace (e.g. `"myorg.eu"` instead of `"rosenpass.eu`) and the label (e.g. `"MyApp Symmetric Encryption"`), the protocol could easily accommodate alternative usage scenarios.

By introducing the concept of protocol extensions, we make this possibility explicit.

1. Reworded the abstract to make it clearer that Rosenpass can be used for other purposes than to secure WireGuard

[6] `https://github.com/rosenpass/rosenpass/pull/664`

2. Reworded Section Symmetric Keys, adding references to the new section on protocol extension
3. Added a `setup_osks()` function in section Hashes, to make the reference to protocol extensions explicit
4. Added a new section on protocol extensions and the standard extension for using Rosenpass with WireGuard
5. Added a new graphic to showcase how Rosenpass and WireGuard interact
6. Minor formatting and intra-document references fixes

2025-05-22 - SHAKE256 keyed hash

Author: David Niehues
PR: #653[7]

We document the support for SHAKE256 with prepended key as an alternative to BLAKE2s with HMAC.

Previously, BLAKE2s with HMAC was the only supported keyed hash function. Recently, SHAKE256 was added as an option. SHAKE256 is used as a keyed hash function by prepending the key to the variable-length data and then evaluating SHAKE256. In order to maintain compatablity without introducing an explcit version number in the protocol messages, SHAKE256 is truncated to 32 bytes. In the update to the whitepaper, we explain where and how SHAKE256 is used. That is:

1. We explain that SHAKE256 or BLAKE2s can be configured to be used on a peer basis.
2. We explain under which circumstances, the reference implementation tries both hash functions for messages in order to determine the correct hash function.
3. We document that the cookie mechanism always uses SHAKE256.

2024-10-30 – InitConf retransmission updates

Author: Karolin Varner
Issue: #331[8]
PR: #513[9]

We redesign the InitConf retransmission mechanism to use a hash table. This avoids the need for the InitConf handling code to account for InitConf retransmission specifically and moves the retransmission logic into less-sensitive code.

Previously, we would specifically account for InitConf retransmission in the InitConf handling code by checking the biscuit number: If the biscuit number was higher than

[7] `https://github.com/rosenpass/rosenpass/pull/653`
[8] `https://github.com/rosenpass/rosenpass/issues/331`
[9] `https://github.com/rosenpass/rosenpass/pull/513`

any previously seen biscuit number, then this must be a new key-exchange being completed; if the biscuit number was exactly the highest seen biscuit number, then the InitConf message is interpreted as an InitConf retransmission; in this case, an entirely new EmptyData (responder confirmation) message was generated as confirmation that InitConf has been received and that the initiator can now cease opportunistic retransmission of InitConf.

This mechanism was a bit brittle; even leading to a very minor but still relevant security issue, necessitating the release of Rosenpass maintenance version 0.2.2 with a fix for the problem[10]. We had processed the InitConf message, correctly identifying that InitConf was a retransmission, but we failed to pass this information on to the rest of the code base, leading to double emission of the same "hey, we have a new cryptographic session key" even if the `outfile` option was used to integrate Rosenpass into some external application. If this event was used anywhere to reset a nonce, then this could have led to a nonce-misuse, although for the use with WireGuard this is not an issue.

By removing all retransmission handling code from the cryptographic protocol, we are taking structural measures to exclude the possibilities of similar issues.

- In section "Dealing With Package Loss" we replace

  > 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 InitConf 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.

  by

  > The responder uses less complex form of the same mechanism: The responder never retransmits RespHello, instead the responder generates a new RespHello message if InitHello is retransmitted. Responder confirmation messages of completed handshake (EmptyData) messages are retransmitted by storing the most recent InitConf messages (or their hashes) and caching the associated EmptyData messages. Through this cache, InitConf retransmission is detected and the associated EmptyData message is retransmitted.

- In function `load_biscuit` we replace

[10] `https://github.com/rosenpass/rosenpass/pull/329`

```
assert(pt.biscuit_no ≤ peer.biscuit_used);
```

by

```
// In December 2024, the InitConf retransmission
 ↪  mechanism was redesigned
// in a backwards-compatible way. See the changelog.
//
// -- 2024-11-30, Karolin Varner
if (protocol_version!(< "0.3.0")) {
    // Ensure that the biscuit is used only once
    assert(pt.biscuit_no ≤ peer.biscuit_used);
}
```

2024-04-16 – Denial of Service Mitigation

Author: Prabhpreet Dua
Issue: #137[11]
PR: #142[12]

- Added denial of service mitigation using the WireGuard cookie mechanism
- 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. 7).

[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 3 Submission. Oct. 2020. https://classic.mceliece.org/nist/mceliece-20201010.pdf (cit. on p. 6).

[11] https://github.com/rosenpass/rosenpass/issues/137
[12] https://github.com/rosenpass/rosenpass/pull/142

[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. 6).

[7] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, Damien Stehlé, et al. *CRYSTALS-Kyber algorithm specifications and supporting documentation*. Tech. rep. Aug. 2021. `https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf` (cit. on p. 6).

[8] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. "Keying hash functions for message authentication". In: *Annual international cryptology conference*. Springer. 1996, pp. 1–15 (cit. on p. 31).

[9] 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 pp. 13, 26).

[10] 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).

[11] Dan Boneh and Victor Shoup. *A graduate course in applied cryptography*. 2023. `https://toc.cryptobook.us/` (cit. on p. 31).

[12] 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, 23).

[13] 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 pp. 1, 31).

[14] 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).

[15] 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).

[16] Trevor Perrin. *The Noise Protocol Framework*. https://noiseprotocol.org/noise.html. July 2018 (cit. on p. 1).

[17] 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 pp. 5, 13).

[18] National Institute of Standards and Technology. *FIPS PUB 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Aug. 2015. DOI: 10.6028/NIST.FIPS.202 (cit. on pp. 5, 15).

[19] 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. 7).