Wire Security Whitepaper

Wire Swiss GmbH*

March 3, 2016

Contents

*privacy@wire.com

1 Introduction

Wire runs on Android and iOS devices, on Windows and OS X as well as on the Web in browsers. Registered users engage in conversations whose contents are synchronized across all devices of a user.

This document provides an overview on the cryptographic protocols and security aspects implemented to protect the privacy of users.

2 Application layer

Wire clients interact with backend servers over HTTPS connections supporting the following TLS parameters:

TLS versions

- TLSv1.0
- TLSv1.1
- TLSv1.2

The server indicates the order preference of cipher suites and communicates HTTP Strict Transport Security [2] to all clients.

In addition to requests to HTTP resources, clients can maintain a websocket connection to receive real-time push notifications, as well as register for push notifications through external transports such as GCM [3] and APNs [4]. See section 5.4 on page 11 for details on push notifications.

3 Registration

Registration on Wire involves up to three steps, whereby only the first is strictly required in order to start using the service:

1. User registration.
2. Client registration.
3. Push token registration.

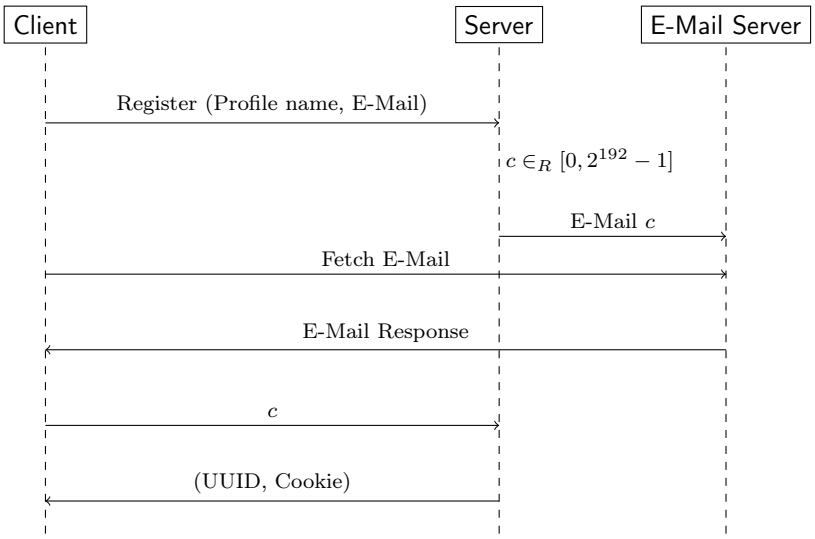

Figure 3.1: User Registration (E-Mail)

3.1 User Registration

Wire supports two basic registration flows, which can optionally be composed. All flows have in common that a profile name must be provided, which does not have to be unique. For more details on the data collected please see the Wire Privacy Whitepaper.

3.1.1 Registration by e-mail

Registration by e-mail (figure 3.1) requires a profile name and a valid e-mail address. To verify the e-mail address, the server generates a random verification code $c \in_R [0, 2^{192} - 1]$ and sends it to the given e-mail address to complete the registration. The server allows only 3 attempts to send the correct verification code before the code is automatically invalidated and a new code needs to be requested. Verification codes expire after two weeks.

Upon successful registration the client receives a Wire internal ID (UUID v4) and an authentication cookie.

3.1.2 Registration by phone

Registration by phone number (figure 3.2) requires a profile name and a valid phone number. Before the client application sends the actual registration request, it asks the server to send a verification code $c \in_R [0, 10^6 - 1]$ via SMS to the phone number the user provided. The actual registration request includes c. A client only has 3 attempts to send the correct verification code before it is invalidated, in which case a new code needs to be requested.

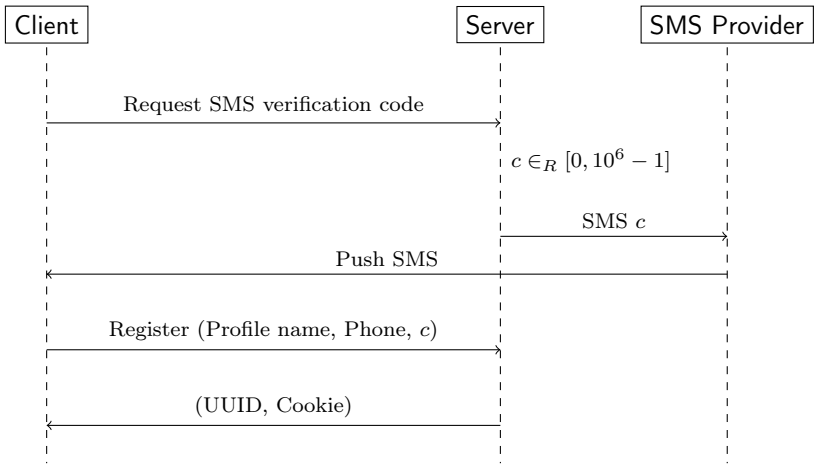

Figure 3.2: User Registration (Phone)

Upon successful registration the client receives a Wire internal ID (UUID v4) and an authentication cookie.

3.1.3 Passwords

Passwords are not stored in plain text on the servers, instead they are passed into the *scrypt* key derivation function [5] with the parameters $N = 2^{14}, r = 8, p = 1$ and a random salt $s \in_R [0, 2^{256} - 1]$. The resulting hashes are stored along with the salt and the parameters in the form $\log_2 N \| r \| p \| base64(s) \| base64(hash)$.

3.1.4 Further data

The following additional data is stored by the servers:

- Locale: An IETF language tag representing the user's preferred language.
- Accent Color: A numeric constant.
- Picture: Metadata about a previously uploaded public profile picture, including a unique ID, dimensions and a tag.
- Cookie Label: A label to associate with the user token that is returned as an HTTP cookie upon successful registration.

3.2 Client registration

Client registration (figure 3.3) is required in order to participate in the exchange of end-to-end encrypted content. The concept of user accounts is less relevant, as encrypted content is exchanged between two clients.

A user can register up to 8 client applications (usually different devices) in total: 7 are *permanent* 1 is *temporary*. Attempts to register more than 7 permanent

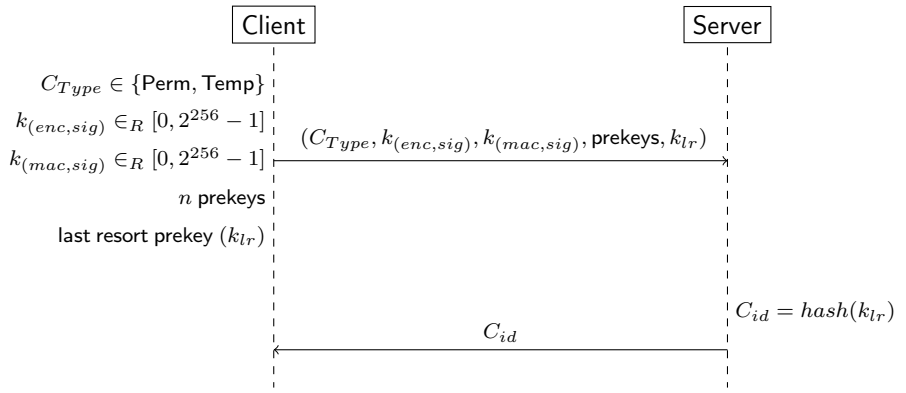

Figure 3.3: Client Registration

clients will result in an error and require permanent client to be removed. Registering a new temporary client will replace the old one.

These restrictions limit the amount of computation clients need to perform when sending encrypted messages, as messages are encrypted individually between clients.

The signaling keys $(k_{(enc,sig)}, k_{(mac,sig)})$ are symmetric keys shared between server and client and are used to encrypt push notifications over external transport chanel in order to minimise metadata exposure. Details on push notifications can be found in section 5.4 on page 11.

The prekeys are used by other clients to initiate cryptographic sessions with the newly registered client and are defined in section 5.1.1 on page 9.

Upon successful client registration the server returns a client ID (C_{id}) which is unique per user ID.

3.2.1 Further data

The following data will also be collected during client registration:

- Class: The device class: *Mobile*, *Tablet* or *Desktop*.
- Model: The device model, e.g. iPhone 4s.
- Label: A human-readable label for the user to distinguish devices of the same class and model.
- Cookie label: A cookie label links the client to authentication cookies (cf. section 4 on the next page). When such a client is later removed from the account, i.e. when a device is lost, the server will revoke any authentication cookies with a matching cookie label. Once set, cookie labels can never be changed.
- Password: If the user has a password, client registration requires reauthentication with this password, with the exception of the first reg-

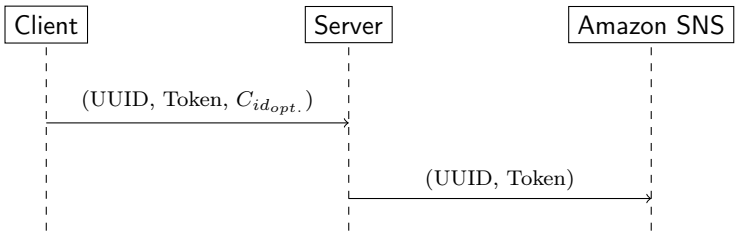

Figure 3.4: Push token registration

istered client of an account. Similarly, removing a registered client also requires the password to be entered.

3.2.2 Metadata

The server collects the following metadata for every newly registered client and makes it available it to the user:

- Timestamp: The UTC timestamp when the client was registered.
- Location: The geo-location of the IP address used to register the client.

This information is only collected to make notifications about new registrations more meaningful.

3.2.3 Notifications

When a new client is registered with an account, all existing clients of the same account are notified of that event. Additionaly, the user will be notified via e-mail and/or SMS. These notifications help the user to identify suspicious clients registered with their account, e.g. when login credentials are stolen.

3.3 Push token registration

As a final registration step a client can register push tokens in order to receive push notifications over GCM or APNs when the device is offline (when there is no open websocket connection). Details about push notifications can be found in section 5.4 on page 11.

4 Authentication

4.1 Tokens

API authentication is based on a combination of short-lived bearer tokens, referred to as *access tokens*, as well as long-lived *user tokens*. Access tokens are

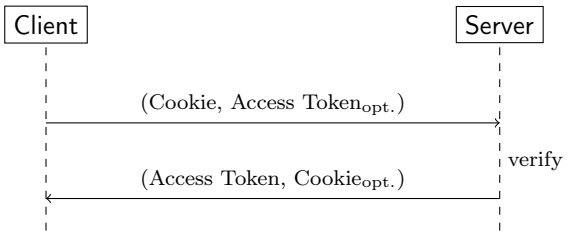

Figure 4.1: Token Refresh

used to authenticate requests to protected API resources and user tokens are used to continuously obtain new access tokens.

User tokens are sent as HTTP cookies. All tokens are strings signed[1] by the server and include the user ID (UUID v4) and the expiration time as a Unix timestamp. The full format of user tokens and access tokens is specified in appendix A on page 13.

The scope of user tokens, and thus cookies, can be *persistent* or *session-based*, with the same semantics as those specified by the HTTP protocol. A client chooses the scope of the cookie during login. The HTTP cookie attributes restrict their use to the domain of the backend server, to the path of the token refresh endpoint as well as to the HTTPS protocol. Persistent cookies are stored in permanent, secure storage on the client, whereas session cookies are kept in ephemeral storage only (e.g. a browser session). Persistent cookies expire after 1 year and session cookies expire after 1 week.

Access tokens are comparatively short-lived (15 minutes). To refresh an expired access token a client uses a cookie to refresh the access token. If the cookie is valid, a new access token is generated and returned. When an access token is refreshed the server may additionally issue a new cookie, thus continuously prolonging the expiration date (figure 4.1). Such a cookie renewal typically occurs approx. every 3 months.

A user may have a maximum of 32 persistent cookies and 32 session cookies, both of which are replaced transparently from least recent to most recent.

After the initial registration, only a user login can generate a new long-lived user token and an access token.

Wire supports two different types of logins described below.

4.2 Login

Users who have added a password to their account or have verified a phone number can login. Logins are classified as *session* or *persistent* logins, which corresponds to the desired scope of the resulting cookie. Clients can choose the type of login.

[1] A cryptographic Ed25519 signature attached to the string.

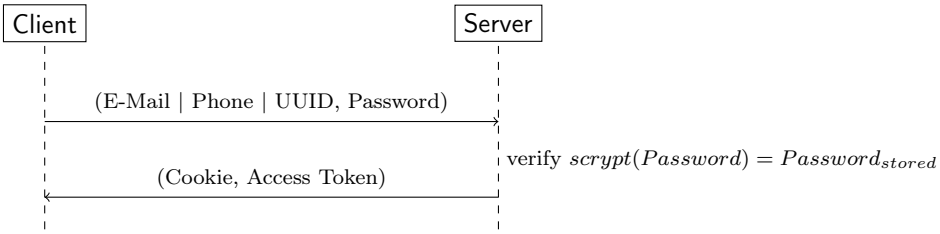

Figure 4.2: Login with password

4.2.1 Password login

To login with a password a client provides a user ID, e-mail address or phone number and a the password, which are transmitted over TLS. The server verifies the password using *scrypt* [5] and issues a new user token as an HTTP cookie and a new access token as shown in figure 4.2.

4.2.2 SMS login

Users who registered with a verified phone number can login via SMS. The procedure is the same as during registration and is subject to the same restrictions, however an SMS login code already expires after 10 minutes.

4.3 Password Reset

Wire provides a self-service password reset [6] for any registered user with a password and a verified e-mail address or phone number.

The procedure for a password reset via a verified phone number or e-mail address is similar to the initial verification (cf. figure 3.1 and 3.2), with the following differences:

- There can be only 1 pending password reset for an account at any time. A new password reset cannot be initiated before the timeout window expires.
- The password reset codes are valid for 10 minutes.

5 Messaging

Messaging refers exchanging text messages and assets (section 5.3). All messaging in Wire is subject to end-to-end encryption to provide users with a strong degree of privacy and security.

5.1 End-to-end encryption

End-to-end encryption (E2EE) takes place between two clients (cf. 3.2). Axolotl [9] is the main cryptographic protocol. It is derived from the Off-the-Record protocol, using a different ratchet[10].

Furthermore Wire uses the concept of prekeys [7] to use the protocol in an asynchronous environment. It is not necessary for two parties to be online at the same time to initiate an encrypted conversation.

The actual Axolotl implementation used in Wire is Proteus [8]. It uses the following cryptographic primitives (provided by libsodium [15]):

- ChaCha20 stream cipher [16]
- HMAC-SHA256 as MAC [17]
- Elliptic curve Diffie-Hellman key exchange (Curve25519 [18])

Key derivation is done using HKDF [19].

5.1.1 Prekeys

Every client initially generates some key material which is stored locally:

- Identity keypair: $(a, g^a) \in_R \mathbb{Z}_p \times \mathsf{Curve25519}$ where $g \in \mathsf{Curve25519}$
- A set of prekeys [7]: $(k_{(a,i)}, g^{k_{(a,i)}}) \in_R \mathbb{Z}_p \times \mathsf{Curve25519}$ where $0 \leq i \leq 65535$.

During client registration (section 3.2) a client uploads prekeys $(g^{k_{(a,0)}}, ..., g^{k_{(a,j)}})$ bundled with its public identity key g^a. These are eventually used by other clients to asynchronously initiate an end-to-end encrypted conversation, i.e. given a recipient's prekey $g^{k_{(a,i)}}$ and identity key g^a the sender can derive n initial encryption key even if the recipient is offline.

The prekey with ID 65535 is the so-called "last resort" prekey. Every prekey is intended to be used only once, which means that the server removes every requested prekey immediately. In order to not run out of prekeys the last resort prekey is never removed and clients should regularly upload fresh prekeys.

For further details on the remaining protocol flow and its security properties please refer to references [9], [10], [11] and [14].

5.2 Message exchange and client discovery

To send an encrypted message the sending client needs to have a cryptographic session with every client it wants to send the message to (usually all clients of all participants of a particular conversation). It will encrypt the plain text message for every recipient and send the batch to the server. The server checks if every client of every user who is a participant of the conversation is part of the batch. If a client is missing, the server will reject the request and inform the sender of

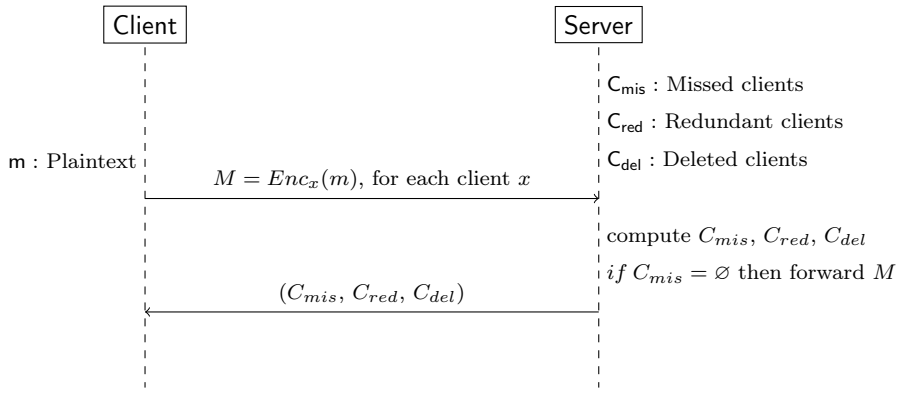

Figure 5.1: Client Discovery (Sender)

missing clients.[2] The sender can then fetch prekeys for the missing clients and prepare the remaining messages before attempting to resend the entire batch.

By the same mechanism clients are also informed about redundant clients, i.e. clients they have prepared an encrypted message for, but which are no longer part of the conversation. This includes deleted clients, i.e. clients which are redundant and known to have been deleted. The sender can use this information to update its own list of clients participating in a conversation and the corresponding cryptographic sessions.

Client discovery for the sender of a message is depicted in figure 5.1.

Conversely, when a client receives an encrypted message from another client with whom no prior cryptographic session exists, it initializes a new cryptographic session from the encrypted message.

To rule out man-in-the-middle attacks users need to compare identity key fingerprints out-of-band.

5.3 Assets

Assets are larger binary entities sent between users, such as pictures.

Profile pictures are uploaded as plaintext assets with technical metadata (e.g. width, height, file type) and are shared through a user's profile.

Any other assets shared in conversations are end-to-end encrypted. Compared to regular text messages, the encryption of assets applies an optimization proposed in [12] to reduce the required computational overhead and network bandwidth for the sender. On Wire, the sending client does the following:

1. It generates a random symmetric key k for use with AES-256.
2. It encrypts the asset data with k using CBC mode with PKCS#5/7 padding and computes the SHA-256 hash of the resulting ciphertext.

[2]Clients do have the ability to override this behaviour, but are always informed about missing clients.

3. It encrypts the key k together with the hash and other asset metadata for each recipient via the Axolotl protocol.
4. It sends the encrypted asset data as well as the encrypted metadata payload for each recipient to the server.

The receiving client of an asset metadata message then does the following:

1. It decrypts the asset metadata using the Axolotl protocol, thus obtaining the symmetric key k as well as the SHA-256 hash of the asset ciphertext.
2. It downloads the asset ciphertext, computes the SHA-256 hash and compares it to the received hash to verify the integrity of the asset data.
3. It decrypts the asset data using the key k.

As with regular text messages, only clients in the same conversation can receive asset metadata messages from one another and are authorized to download the corresponding asset ciphertext.

Assets are persistently stored on the server without a predefined timeout. This means that a client can repeatedly download and decrypt the same asset to conserve disk space on the device, since the client persistently stores the decrypted symmetric key k together with the SHA-256 hash. These credentials have the same sensitivity as the plaintext asset itself. Forward secrecy is not affected since the decryption key k is sent using the Axolotl protocol.

5.4 Notifications

Messages are delivered by the server to recipients via notifications. Notifications are delivered by Wire over 3 different channels.

Websocket connections: Every authenticated client can establish a websocket connection over HTTPS. A client with an established websocket connection is considered *online*.

External push providers: Wire currently supports GCM and APNs as external push providers. This channel is used if a client is offline but has registered a valid GCM or APNs push token with the server. The content is encrypted and not visible to the external push providers.

Notification queues: Every message sent by a user, as well as most metadata messages are enqueued in a per-client notification queue that can be queried (and filtered) by every registered, authenticated client of a user. The notification queue allows clients to retrieve messages they may have missed. The retention period of notifications is 4 weeks.

Notifications for end-to-end encrypted messages that are sent to external push providers are additionally encrypted using the signaling keys (cf. section 3.2) in order to protect sender information and other metadata accompanying the encrypted message. For that purpose the following cryptographic primitives are used in an encrypt-then-MAC scheme:

- HMAC-SHA256 [17]

- AES256 [20] in CBC mode with PKCS#5/7 padding. The IV consists of 16 random bytes (the block length) and is prepended to the ciphertext.

6 Calling

Wire users can call each other in 1:1 or group conversations. Calls are initiated to all participants of a conversation and users who are not a participant of that conversation do not have access to the call. Group calls are currently limited to 5 active participants, and are offered only in conversations with no more than 10 participants. These limits may change over time.

The codec used for streaming audio is Opus.

Setting up a call involves three aspects: signaling, media transport and encryption. These are described in detail next.

Known limitation: Currently calling doesn't use the same cryptographic identity as text messages and assets. This limitation will be addressed in future releases.

6.1 Call signaling

Call signaling establishes a connection between clients and negotiates their common capabilities by exchanging SDP messages. During this phase, the clients communicate through a server component via HTTPS requests on the uplink and websockets on the downlink.

During the call, clients also send messages at regular intervals to the server component, to inform that the call is still ongoing.

6.2 Media transport

Once connected, endpoints determine a transport path for the media between them. Whenever possible the endpoints allow direct media flow between them, however some networks may have firewalls or NATs preventing direct streaming and instead require the media to be relayed through a TURN server. ICE identifies the most suitable transport path.

6.3 Encryption

Call media is exchanged between endpoints in an SRTP-encrypted media session. To initiate the session the SRTP encryption algorithm, keys, and parameters are negotiated through a DTLS handshake. The authenticity of the clients is also verified during the handshake.

In a group call, every participant connects to every other participant as if they were in a 1:1 call. Therefore, all legs of the group call are individually encrypted and encryption keys are not shared among participants.

6.4 WebRTC

Wire is fully compliant with WebRTC and existing IETF standards. As a result, Wire native endpoints can also securely exchange media with any WebRTC compliant web-browser such as Google Chrome or Mozilla Firefox.

These are the main IETF standards used by Wire:

- UDP (RFC 768[21])
- RTP (RFC 3550[22])
- ICE (RFC 5245[23])
- STUN (RFC 7350[24])
- TURN (RFC 5766[25])
- SDP (RFC 4566[26])
- SRTP (RFC 3711[27])
- DTLS (RFC 4347[28])
- DTLS-SRTP (RFC 5764[29])
- Opus (RFC 6716[30]).

Appendices

A Cookie and access token format

⟨*token*⟩ ::= ⟨*signature*⟩
‘`.v=`’ ⟨*version*⟩
‘`.k=`’ ⟨*key-index*⟩
‘`.d=`’ ⟨*timestamp*⟩
‘`.t=`’ ⟨*type*⟩
‘`.l=`’ ⟨*tag*⟩
‘`.`’ ⟨*type-specific-data*⟩

⟨*version*⟩ ::= ⟨*Integer*⟩

⟨*key-index*⟩ ::= ⟨*Integer*⟩

⟨*timestamp*⟩ ::= ⟨*Integer*⟩

⟨*type*⟩ ::= ‘`a`’ | ‘`u`’

⟨*tag*⟩ ::= ‘`s`’ | ‘’

⟨*type-specific-data*⟩ ::= ‘`a=`’ ⟨*access-data*⟩ | ‘`u=`’ ⟨*user-data*⟩

⟨*access-data*⟩ ::= ⟨*UUID*⟩ ‘`.c=`’ ⟨*Word64*⟩

⟨*user-data*⟩ ::= ⟨*UUID*⟩ ‘`.r=`’ ⟨*Word32*⟩

References

[1] http://electron.atom.io

[2] https://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security

[3] https://developers.google.com/cloud-messaging/

[4] https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Chapters/ApplePushService.html

[5] http://www.tarsnap.com/scrypt.html

[6] https://en.wikipedia.org/wiki/Self-service_password_reset

[7] https://whispersystems.org/blog/asynchronous-security/

[8] https://github.com/wireapp/proteus

[9] https://github.com/trevp/axolotl/wiki

[10] https://whispersystems.org/blog/advanced-ratcheting/

[11] https://whispersystems.org/blog/simplifying-otr-deniability/

[12] https://whispersystems.org/blog/private-groups/

[13] https://github.com/WhisperSystems/Signal-Android

[14] https://eprint.iacr.org/2014/904.pdf

[15] https://github.com/jedisct1/libsodium

[16] https://en.wikipedia.org/wiki/Salsa20#ChaCha_variant

[17] https://en.wikipedia.org/wiki/Hash-based_message_authentication_code

[18] https://en.wikipedia.org/wiki/Curve25519

[19] https://tools.ietf.org/html/rfc5869

[20] https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[21] http://tools.ietf.org/html/rfc768

[22] http://tools.ietf.org/html/rfc3550

[23] https://tools.ietf.org/html/rfc5245

[24] https://tools.ietf.org/html/rfc5389

[25] https://tools.ietf.org/html/rfc5766

[26] https://tools.ietf.org/html/rfc4566

[27] https://tools.ietf.org/html/rfc3711

[28] https://tools.ietf.org/html/rfc4347

[29] http://tools.ietf.org/html/rfc5764

[30] https://tools.ietf.org/html/rfc6716