# Signal Protocol Implementation in Python for End-to-End Encryption

Mini project for CS4035D (Computer Security) in Semester 6

Team No : 11

Team Members:

Alen Antony (B200735CS)

Abhiram J (B200733CS)

Vivek Pankaj (B200757CS)

# Index

1. Abstract
2. Introduction
3. Literature Survey
4. System requirements
5. Design
6. Progress of implementation
7. Conclusion
8. References

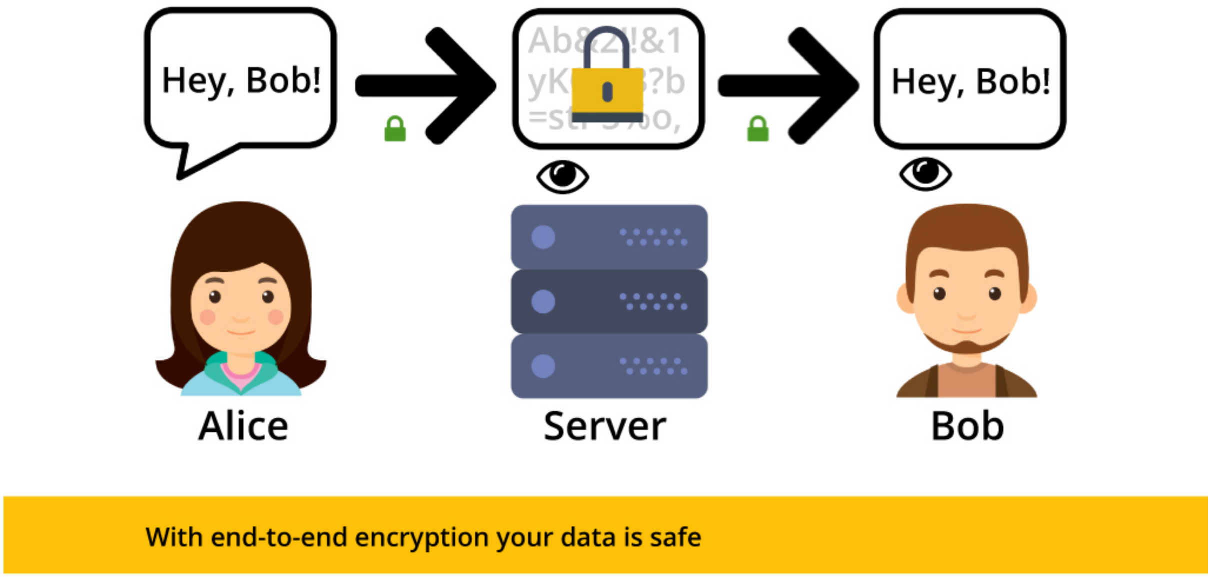
## Abstract

The project is an implementation of the Signal end-to-end encryption protocol in Python. The implementation includes all member functions such as the Extended Triple Diffie Hellman Key Exchange and Double Ratchets. The project aims to provide a secure and private communication channel between two parties. The Signal protocol is known for its strong security guarantees and has been adopted by many popular messaging apps. The implementation of the Signal protocol in Python will allow developers to integrate end-to-end encryption into their own applications.

## Introduction

In today's digital age, communication is essential, and people are increasingly relying on messaging systems to communicate with each other. However, with the rise of cyber threats and privacy concerns, it has become crucial to ensure that communication is secure and private. An end-to-end encrypted messaging system can provide an effective solution to these concerns.

Our project is to design and develop an end-to-end encrypted messaging system that allows users to communicate securely and privately. The system will use the signal protocol to ensure that all messages exchanged between users are end to end encrypted.



## Literature Survey

The Extended Triple Diffie Hellman (X3DH) key agreement protocol is used in the Signal Protocol to establish a shared secret key which may be used for bidirectional communication. The protocol involves three parties: Alice, Bob, and a server. During each protocol run, Alice generates a new ephemeral key pair with public key EK A. After a successful protocol run, Alice and Bob will share a 32-byte secret key SK. This key may be used within some post-X3DH secure communication protocol.

An alternative to RSA, ECC provides the same level of security with a 256-bit key that RSA provides with a 3072-bit key.

The Extended Edwards Curve Digital Signature Algorithm (XEdDSA) and its verifiable random function (VXEdDSA) signature schemes are used in the Signal Protocol to enable the use of a single key pair format for both elliptic curve Diffie-Hellman and signatures. XEdDSA enables using the same key pair for both algorithms in some situations. We are using curve 25519. Curve25519 uses a

Montgomery curve over the prime field defined by the prime number 2^255 − 19. The curve equation is $y^2 = x^3 + 486662x^2 + x$.

Some alternatives are

1. NIST P-256 : The curve equation for NIST P-256 is $y^2 = x^3 - 3x + b \bmod q$, where q is a prime number that defines the finite field over which the curve is defined.
2. Curve448: Curve448 is an untwisted Edwards curve with the equation: $y^2 + x^2 = 1 - 39081x^2y^2$. It allows fast performance compared with other proposed curves with comparable security.

The Double Ratchet algorithm provides forward secrecy by using a new key for each message. The key is derived from the previous key and some random data. This means that if an attacker gains access to one key, they will not be able to decrypt any other messages. The Double Ratchet algorithm also provides backward secrecy, which means that if an attacker gains access to a message, they will not be able to decrypt any previous messages.

| No. | Tools/Techniques | Functionality | Links/References |
|---|---|---|---|
| 1. | Extended Triple Diffie Hellman (X3DH) | Establishes shared secret keys with forward secrecy. Also, the key is established asynchronously using the server. | Signal >> Specifications >> The X3DH Key Agreement Protocol<br><br>https://whispersystems.org/docs/specifications/x3dh/ |

|  |  |  |  |
|---|---|---|---|
| 2. | XEdDSA and VXEdDSA | Signature schemes for twisted Edwards Curves and equivalent Montgomery Curves. Here we are using curve 25519. | https://whispersystems.org/docs/specifications/xeddsa/ |
| 3. | Double Ratchet |  | Signal >> Specifications >> The Double Ratchet Algorithm |

## System Requirements

A personal computer with 2gb RAM and python >= 3.8 installed.

The following python packages are required:

- cryptography
- pycryptodome
- python-axolotl-curve25519

## Design

The two main modules of the project are the **X3DH key agreement protocol** and the **Double Ratchet Algorithm**.

### X3DH

The "X3DH" (or "Extended Triple Diffie-Hellman") key agreement protocol establishes a shared secret key between two parties who mutually authenticate each other based on public keys. X3DH provides forward secrecy and cryptographic deniability.

X3DH is designed for asynchronous settings where one user ("Bob") is offline but has published some information to a server. Another user ("Alice") wants to use that information to send encrypted data to Bob, and also establish a shared secret key for future communication.

For the implementation of the algorithm, we will be using the curve X25519 and the SHA-256 hash function.

## Notation

- The concatenation of byte sequences ***X*** and ***Y*** is ***X*** || ***Y***.
- ***DH(PK1, PK2)*** represents a byte sequence which is the shared secret output from an Elliptic Curve Diffie-Hellman function involving the key pairs represented by public keys *PK1* and *PK2*.
- ***Sig(PK, M)*** represents a byte sequence that is an XEdDSA signature on the byte sequence *M* and verifies with public key *PK*, and which was created by signing *M* with *PK*'s corresponding private key.
- ***KDF(KM)*** represents 32 bytes of output from the HKDF algorithm which takes secret key material formed from the Diffie-Hellman key exchange as the input along with the salt.

## Roles

The X3DH protocol involves three parties: Alice, Bob, and a server.

- **Alice** wants to send Bob some initial data using encryption, and also establish a shared secret key which may be used for bidirectional communication.
- **Bob** wants to allow parties like Alice to establish a shared key with him and send encrypted data. However, Bob might be offline when Alice attempts to do this. To enable this, Bob has a relationship with some server

- The **server** can store messages from Alice to Bob which Bob can later retrieve. The server also lets Bob publish some data which the server will provide to parties like Alice

## Keys

X3DH uses the following elliptic curve public keys:

- IKa - Alice's identity key
- EKa - Alice's ephemeral key
- IKb - Bob's identity key
- SPKb - Bob's signed prekey
- OPkb - Bob's one-time prekey

All public keys will have their respective private keys.

Each client has a long-term identity public key (IKa for Alice, IKb for Bob).

Bob also has a signed pre-key SPKb, which he will change periodically, and a set of one-time pre-keys OPKb, which are used in a single X3DH protocol run.

During each protocol run, Alice generates a new ephemeral key pair with public key Eka.

After a successful protocol run Alice and Bob will share a 32-byte secret key SK. This key is used as the input to the Double Ratchet algorithm.

## The X3DH protocol

X3DH has three phases:

1. Bob publishes his identity key and prekeys to a server.
2. Alice fetches a "prekey bundle" from the server, and uses it to send an initial message to Bob.
3. Bob receives and processes Alice's initial message.

The following sections explain these phases.

## <u>Publishing keys</u>

Bob publishes a set of elliptic curve public keys to the server, containing:

- Bob's identity key *IKb*
- Bob's signed prekey *SPKb*
- Bob's prekey signature *Sig(IKb,Encode(SPKb))*
- A set of Bob's one-time prekeys *(OPKb1,OPKb2,OPKb3,...)*

Bob only needs to upload his identity key to the server once. However, Bob may upload new one-time prekeys at other times. Bob will also upload a new signed prekey and prekey signature at some interval. The new signed prekey and prekey signature will replace the previous values. After uploading a new signed prekey, Bob may keep the private key corresponding to the previous signed prekey around for some period of time, to handle messages using it that have been delayed in transit. Eventually, Bob should delete this private key for forward secrecy.

### Sending the initial message

To perform an X3DH key agreement with Bob, Alice contacts the server and fetches a "prekey bundle" containing the following values:

- Bob's identity key *IKb*
- Bob's signed prekey *SPKb*
- Bob's prekey signature *Sig(IKb, Encode(SPKb))*
- (Optionally) Bob's one-time prekey *OPKb*

The server should provide one of Bob's one-time prekeys if one exists, and then delete it. If all of Bob's one-time prekeys on the server have been deleted, the bundle will not contain a one-time prekey.

Alice verifies the prekey signature and aborts the protocol if verification fails. Alice then generates an ephemeral key pair with public key *EKa*.

If the bundle does not contain a one-time prekey, she calculates:

DH1 = DH(IKa, SPKb)

DH2 = DH(EKa, IKb)

DH3 = DH(EKa, SPKb)

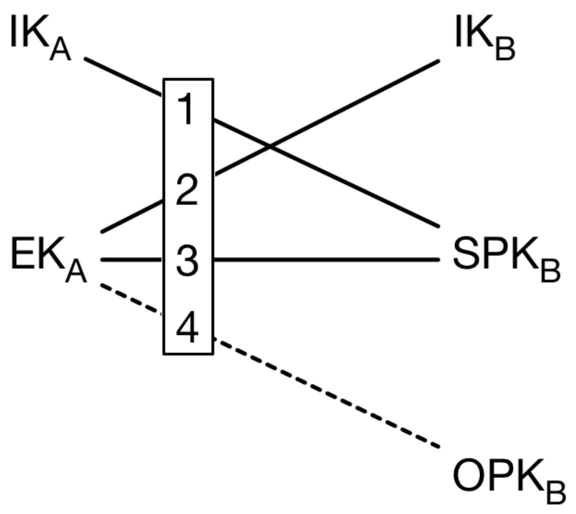
SK = KDF(DH1 || DH2 || DH3)

If the bundle *does* contain a one-time prekey, the calculation is modified to include an additional *DH*:

DH4 = DH(EKa, OPKb)

SK = KDF(DH1 || DH2 || DH3 || DH4)

The following diagram shows the *DH* calculations between keys. Note that *DH1* and *DH2* provide mutual authentication, while *DH3* and *DH4* provide forward secrecy.



After calculating *SK*, Alice deletes her ephemeral private key and the *DH* outputs.

Alice then calculates an "associated data" byte sequence *AD* that contains identity information for both parties:

AD = Encode(IKa) || Encode(IKb)

Alice may optionally append additional information to *AD*, such as Alice and Bob's usernames, certificates, or other identifying information.

Alice then sends Bob an initial message containing:

- Alice's identity key IKa
- Alice's ephemeral key EKa
- Identifiers stating which of Bob's prekeys Alice used
- An initial ciphertext encrypted with some AEAD encryption scheme (AES) using *AD* as associated data and using an encryption key which is SK.

The initial ciphertext is typically the first message in some post-X3DH communication protocol (Double Ratchet). In other words, this ciphertext typically has two roles, serving as the first message within some post-X3DH protocol, and as part of Alice's X3DH initial message.

After sending this, Alice may continue using *SK* or keys derived from *SK* within the post-X3DH protocol for communication with Bob.

## <u>**Receiving the initial message**</u>

Upon receiving Alice's initial message, Bob retrieves Alice's identity key and ephemeral key from the message. Bob also loads his identity private key, and the private key(s) corresponding to whichever signed prekey and one-time prekey (if any) Alice used.

Using these keys, Bob repeats the *DH* and *KDF* calculations from the previous section to derive *SK*, and then deletes the *DH* values.

Bob then constructs the *AD* byte sequence using *IKa* and *IKb*, as described in the previous section. Finally, Bob attempts to decrypt the initial ciphertext using *SK* and *AD*. If the initial ciphertext fails to decrypt, then Bob aborts the protocol and deletes *SK*.

If the initial ciphertext decrypts successfully the protocol is complete for Bob. Bob deletes any one-time prekey private key that was used, for forward secrecy. Bob may then continue using *SK* or keys derived from *SK* within the post-X3DH protocol for communication with Alice.

## Double Ratchet

The Double Ratchet algorithm is used by two parties to exchange encrypted messages based on a shared secret key. The parties derive new keys for every Double ratchet message so that earlier keys cannot be calculated from later ones. The parties also send Diffie-Hellman public values attached to their messages .The results of Diffie Hellman calculations are mixed into derived keys so that the later keys cannot be calculated from earlier ones. These properties give some protection to earlier or later messages in case of a compromise of a party's keys.

## KDF chains

We define a **KDF** as a cryptographic function that takes a secret and random **KDF key** and some input data and returns output data. The output data is indistinguishable from random provided the key isn’t known. If the key is not secret and random, the KDF should still provide a secure cryptographic hash of its key and input data. The HMAC and HKDF constructions, when instantiated with a secure hash algorithm, meet the KDF definition.

We use the term **KDF chain** when some of the output from a KDF is used as an **output key** and some is used to replace the KDF key, which can then be used with another input.

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

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

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

## Symmetric-key ratchet

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

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

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

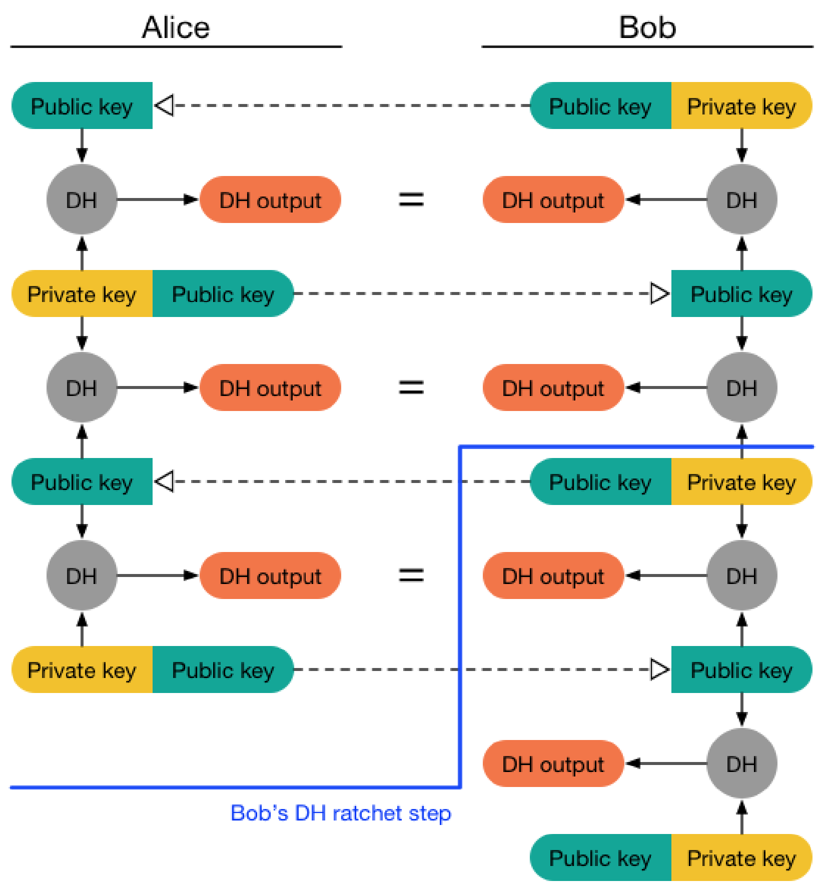
## Diffie-Hellman ratchet

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

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

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

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



## Double Ratchet

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

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

## Double Ratchet Implementation

### External functions

To instantiate the Double Ratchet requires defining the following functions.

- **GENERATE_DH()**: Returns a new Diffie-Hellman key pair.
- **DH(dh_pair, dh_pub)**: Returns the output from the Diffie-Hellman calculation between the private key from the DH key pair dh_pair and the DH public key dh_pub. If the DH function rejects invalid public keys, then this function may raise an exception which terminates processing.
- **KDF_RK(rk, dh_out)**: Returns a pair (32-byte root key, 32-byte chain key) as the output of applying a KDF keyed by a 32-byte root key rk to a Diffie-Hellman output dh_out.
- **KDF_CK(ck)**: Returns a pair (32-byte chain key, 32-byte message key) as the output of applying a KDF keyed by a 32-byte chain key ck to some constant.
- **ENCRYPT(mk, plaintext, associated_data)**: Returns an AEAD encryption of plaintext with message key mk. The associated_data is authenticated but is not included in the ciphertext. Because each message key is only used once, the AEAD nonce may be handled in several ways: fixed to a constant; derived from mk alongside an independent AEAD encryption key; derived as an additional output from KDF_CK(); or chosen randomly and transmitted.
- **DECRYPT(mk, ciphertext, associated_data)**: Returns the AEAD decryption of ciphertext with message key mk. If authentication fails, an exception will be raised that terminates processing.

- **HEADER(dh_pair, pn, n)**: Creates a new message header containing the DH ratchet public key from the key pair in dh_pair, the previous chain length pn, and the message number n. The returned header object contains ratchet public key dh and integers pn and n.
- **CONCAT(ad, header)**: Encodes a message header into a parsable byte sequence, prepends the ad byte sequence, and returns the result. If ad is not guaranteed to be a parsable byte sequence, a length value should be prepended to the output to ensure that the output is parsable as a unique pair (ad, header).

## State variables

The following state variables are tracked by each party:

- **DHs** : DH Ratchet key pair (the "sending" or "self" ratchet key)
- **DHr** : DH Ratchet public key (the "received" or "remote" key)
- **RK** : 32-byte Root Key
- **CKs, CKr** : 32-byte Chain Keys for sending and receiving
- **Ns, Nr** : Message numbers for sending and receiving
- **PN** : Number of messages in previous sending chain

## Initialization

Prior to initialization both parties must use some key agreement protocol (X3DH) to agree on a 32-byte shared secret key SK and Bob's ratchet public key. These values will be used to populate Alice's sending chain key and Bob's root key. Bob's chain keys and Alice's receiving chain key will be left empty, since they are populated by each party's first DH ratchet step.

Once Alice and Bob have agreed on SK and Bob's ratchet public key, Alice and Bob calls RatchetInit() which initializes the ratchet.

## **Encrypting messages**

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

## **Decrypting messages**

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

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

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

### <u>**Integration with X3DH**</u>

The Double Ratchet algorithm can be used in combination with the X3DH key agreement protocol. The Double Ratchet plays the role of a "post-X3DH" protocol which takes the session key SK negotiated by X3DH and uses it as the Double Ratchet's initial root key.

The following outputs from X3DH are used by the Double Ratchet:

- The SK output from X3DH becomes the SK input to Double Ratchet initialization
- The AD output from X3DH becomes the AD input to Double Ratchet encryption and decryption
- Bob's signed prekey from X3DH (SPKB) becomes Bob's initial ratchet public key (and corresponding key pair) for Double Ratchet initialization.

Any Double Ratchet message encrypted using Alice's initial sending chain can serve as an "initial ciphertext" for X3DH. To deal with the possibility of lost or out-of-order messages, a recommended pattern is for Alice to repeatedly send the same X3DH initial message prepended to all her Double Ratchet messages until she receives Bob's first Double Ratchet response message.

## <u>**Progress of implementation**</u>

A simple version of X3DH is implemented. Work is being done on the implementation of Double Ratchet.

## Conclusion

In conclusion, our end-to-end encrypted messaging system will provide users with a secure and private way to communicate with each other. This makes it an ideal messaging solution for individuals and businesses alike. With our system, users can rest assured that their messages and data are fully protected.

Through implementing this project we learned how to make a secure channel for communication, the various protocols and methods used for this and the challenges faced during implementation.

It is worth noticing that Elliptic Curve Cryptography is vulnerable to powerful Quantum computers and therefore it is a possibility that this system might need improvement with new and powerful cryptographic techniques in the future as the power of computers keep on rising.

## References

- Moxie Marlinspike, Trevor Perrin

  “The X3DH Key Agreement Protocol ” signal.org
  https://signal.org/docs/specifications/x3dh/
  (accessed Mar, 24 2023)

- Moxie Marlinspike, Trevor Perrin

  “The Double Ratchet Algorithm ” signal.org
  https://signal.org/docs/specifications/doubleratchet/
  (accessed Mar, 24 2023)