POLITECNICO DI TORINO

**Corso di Laurea Magistrale in Ingegneria Matematica**

Tesi di Laurea Magistrale

# WhatsApp: cryptographic aspects

**Relatore**
Prof. Antonio J. Di Scala

**Candidato**
Andrea Gangemi

*Ottobre 2019*

*Alla mia famiglia, che mi ha sostenuto.*
*A Francesca, Giulio, Marco e Margherita, che mi hanno cambiato.*

# Contents

# Chapter 1

# Introduction

In the last years, instant messaging applications substituted classic SMS texts. At first, messages were not encrypted, or they actually were but with unreliable algorithms. After the mass surveillance scandal of 2013, security became a must for every messaging application. The first applications in commerce with encrypted messages were Threema and Telegram; WhatsApp took some time before transictioning to a system with reliable security. In 2014, WhatsApp signed a partnership with Open Whisper Systems, a no profit group which was developing a protocol for secure instant messaging, called Signal; the integration was completed in the April of 2016, when both companies claimed end-to-end encryption and a way to verify user keys were now available on the app. This means that, starting from that month, all messages , phone calls, videos, or any other form of information exchanged could not be read by any unauthorized entity, WhatsApp included.
Aim of this thesis is an accurate description of the WhatsApp end-to-end encryption protocol, from both mathematical and cryptographic points of view.
Chapter 2 is crypto-oriented and it recaps the main cryptographic algorithms developed during the last years which are used to encrypt and authenticate strings of data.
Chapter 3 is instead math-oriented and it is incentrated on elliptic curves, a modern mathematical instrument which has found various applications in cryptography thanks to the advantage they have over other classic structures in the generation of secure key pairs. Montgomery curves will be the focus on this part, because of an algorithm known as Montgomery Ladder which speeds up the counting process.

Chapter 4 faces the main argument of the thesis, focussing on the end-to-end encryption protocol. It will be described in detail, starting from the moment where an user installs the application, until when she regularly exchanges messages. Informations about the security for both direct messaging and multi-device communications, as well as some details about a possible implementation of the protocol are provided.
Lastly, Chapter 5 mainly talks about the additional WhatsApp features, starting from well known and widely utilized ones like group chats or phone calls, and ending with some niche features like live locations.

# Chapter 2

# Short review of basic cryptography

In this first Chapter, we briefly recap some of the basic cryptographic algorithms which will be later used by the Signal protocol to guarantee end-to-end encryption. In particular, in Section 2.1 we talk about symmetric cryptography with a detailed description of AES and the CBC mode, in Section 2.2 we describe the Hash function SHA-2, in Section 2.3 we define the Message Authentication Code and its application in the generation of new, stronger cryptografic keys (HKDF), in Section 2.4 we put together all what we have learned in the first part of the Chapter to illustrate an algorithm which guarantees confidentiality, integrity, authentication (AEAD). Finally, in Section 2.5 we summarise the role of a digital signature in cryptography.

## 2.1 Block ciphers: AES

In symmetric cryptography, there are two kinds of ciphers: stream ciphers and block ciphers. In a stream cipher, every bit of the message $M$ is XORed with a new, unpredictable, bit, produced by a Pseudo Random Number Generator (PRNG). Instead, for a block cipher, $M$ is divided into blocks of the same length, i.e. $M = B_N||\dots B_2||B_1$, where "$||$" denotes the concatenation of bytes/bits sequences: if the last block has less bits than the necessary, we proceed with a padding. To encrypt a block $B_i$ with $b$ bits we use a key $k$, obtaining the cipher block $Y = E_k(B_i)$. Usually, $Y$ has $b$ bits too. $Y$ is obtained applying multiple times the same functions to give confusion and diffusion to the ciphertext. We call *round* $R$ one single iteration of these functions. The key $k$ can have more bits than the ones of the block: in this case, a key schedule algorithm gives a different key $k_i$, with the desired length, for every round $R_i$.

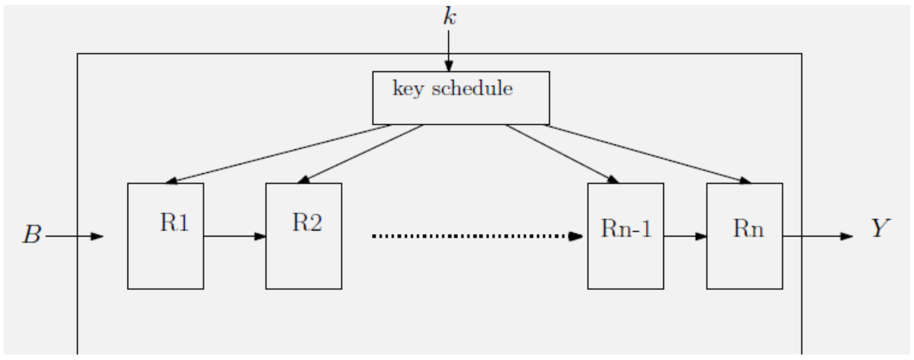


Figure 2.1: Rounds of a block cipher

The *Advanced Encryption Standard* (AES) is the most widely used symmetric block cipher. It is byte oriented. AES takes blocks with length 128 bits and accepts keys with length 128, 192 and 256 bits. The number of internal rounds of the cipher is a function of the key length: there are 10, 12 and 14 rounds respectively. AES encrypts all 128 bits in one iteration: this explains why it has a small number of rounds. AES consists of 3 different "layers". Each layer manipulates all 128 bits of the state $A$ of the algorithm. Each round, with the exception of the first which only performs the first layer, consists of all three layers. Let's give some notation: we call $x$ the plaintext, $y$ the ciphertext and $n_r$ the number of rounds. The state $A$ is arranged in a $4 \times 4$ matrix where each position contains a single byte:

$$A = \begin{bmatrix} A_0 & A_4 & A_8 & A_{12} \\ A_1 & A_5 & A_9 & A_{13} \\ A_2 & A_6 & A_{10} & A_{14} \\ A_3 & A_7 & A_{11} & A_{15} \end{bmatrix}$$

The 3 layers are the following:

- **Key Addition layer**: A 128-bits round key, or a subkey derived from the main key in the key schedule, is XORed to the state. The key bytes are arranged into a matrix with four rows and four (128-bits key), six (192-bits key) or eight (256-bits key) columns, where each element contains a single byte. An XOR addition of a subkey is used both at the input and output of AES: this process is called key whitening. The number of subkeys is equal to the number of rounds plus one, due to the key needed for key whitening in the first key addition layer. The AES subkeys are computed recursively, i.e., in order to derive subkey $k_i$, subkey $k_{i-1}$ must be known, and so on. The AES key schedule is word-oriented, where 1 word = 32 bits. Subkeys are stored in a key expansion array $W$ that consists of words. There are different key schedules for the three different AES key sizes of 128, 192 and 256 bits. WhatsApp uses AES256, so let's describe its key schedule.

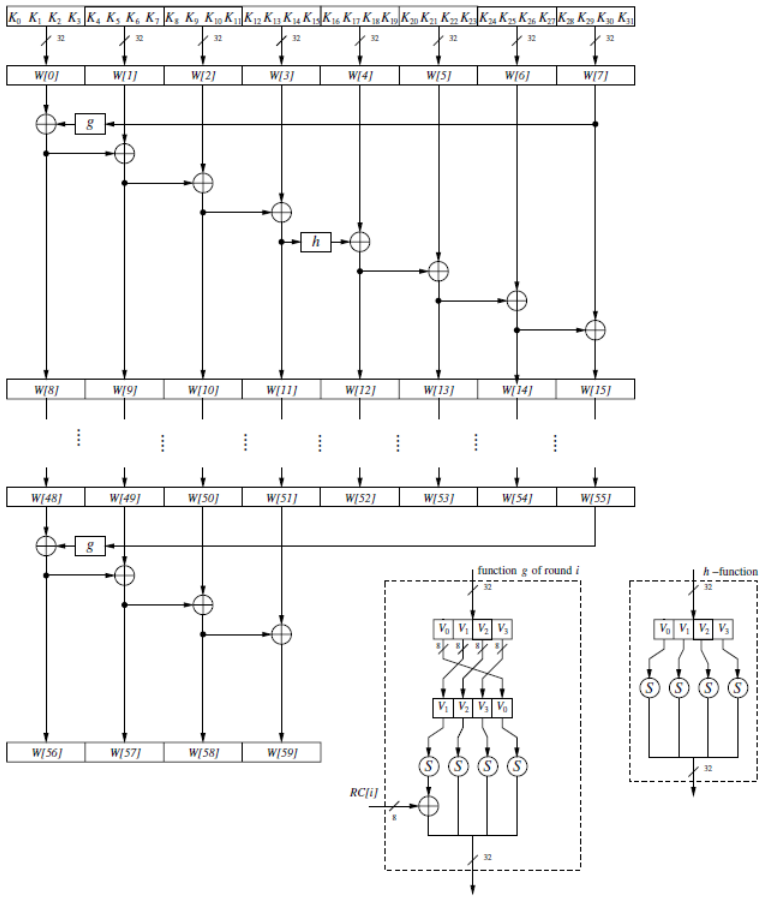


Figure 2.2: AES key schedule for 256-bits key size

AES with 256-bits key needs 15 subkeys. Define the components of the key matrix as $K_0, \ldots, K_{31}$. The subkeys are stored in the 60 words

$W[0], \dots, W[59]$. The computation of the array elements is shown in Figure 2.2. The key schedule has 7 iterations, where each iteration computes 8 words for the subkeys. The subkey for the first AES round is formed by the array elements $(W[0], W[1], W[2], W[3])$, the second subkey by the elements $(W[4], W[5], W[6], W[7])$, and so on. There are seven round coefficients $RC[1], ..., RC[7]$ within the function $g()$. The function $g()$ rotates its four input bytes, performs a byte-wise S-Box substitution, and adds a round coefficient $RC$ to it. $g()$ has two purposes: it adds nonlinearity to the key schedule and removes symmetry in AES. This key schedule also has a function $h()$ with 4-bytes input and output. The function applies the S-Box to all four input bytes. More details of the key schedule algorithm can be found in [1].

- **Byte Substitution layer (S-Box)** : each element of the state gets a non-linear trasformation. This introduces confusion to the data. Computations are done in the field $F_2[x]/m(x)$, with $m(x) = x^8 + x^4 + x^3 + x + 1$, which is an irreducible polynomial on $F_2$. A byte $a = (a_7 a_6 \dots a_0)$ is identified from the polynomial $a_7x^7+a_6x^6+\cdots+a_1x+a_0$. Operations on the byte are computed modulo the polynomial $m(x)$. S-Box is a bijective function $S : \mathbb{Z}_2^8 \to \mathbb{Z}_2^8$ such that $S(a) = Aa^{-1} + v$, where $A$ and $v$ are a fixed matrix and a fixed vector. Their definition can be found, for example, in [3]. $a^{-1}$ can be computed efficiently with the Extended Euclidean Algorithm.

- **Diffusion layer**: as the name suggests, this layer provides diffusion to the data. It is a combination of two sub-layers, both with only linear trasformations:

  - **ShiftRows**: in this sub-layer, the bytes in every row of the state matrix $A$ are shifted in the following way: no shift for the first row, $<<<_1$ for the second row, $<<<_2$ for the third row, $<<<_3$ for the last row. The symbol $<<<_n$ means we do a left rotation of $n$ positions.

  - **MixColumns**: it is a matrix operation on every column of $A$, which combines blocks of 4 bytes. The operation is the following:

$$\begin{bmatrix}\alpha'_0\\\alpha'_1\\\alpha'_2\\\alpha'_3\end{bmatrix}=\begin{bmatrix}02&03&01&01\\01&02&03&01\\01&01&02&03\\03&01&01&02\end{bmatrix}\begin{bmatrix}\alpha_0\\\alpha_1\\\alpha_2\\\alpha_3\end{bmatrix}$$

where $(\alpha_0, \alpha_1, \alpha_2, \alpha_3)$ is a column of $A$ and coefficients in the matrix are hexadecimal numbers which represent elements of $F_{256}$. If we represent these bytes with the polynomial $\alpha(x) = \alpha_3x^3 + \alpha_2x^2 + \alpha_1x + a_0$, $\alpha_i \in F_{256}$ we can also see this passage as the multiplication of $\alpha(x)$ with the fixed polynomial $c(x) = (03)_{16}x^3 + (01)_{16}x^2 + (01)_{16}x + (02)_{16} \mod x^4 + 1$.

The last round $n_r$ does not make use of the MixColumn transformation. Every round of AES is invertible because every function is invertible. To decrypt a message encrypted with AES, we must reverse the order of the operations done for encryption, using the same keys but in reverse order. The layers S-Box, ShiftRows and MixColumns must be substituted with their inverse. The definition of the inverse functions and more details about AES security can be found in [1].

We have seen how AES encrypts a single block. How can we encrypt the whole message? Different "modes" can be used. The mode chosen from WhatsApp is called *Cipher Block Chaining* (CBC). It provides confidentiality for a message sent from A to B. There are two main ideas behind CBC mode. First, the ciphertext $y_i$ depends not only on block $B_i$ but on all previous plaintext blocks $B_{i-1}, \dots, B_1$. Second, the encryption is randomized by using an Initialization Vector (IV): this is an example of probabilistic encryption, if we encrypt two blocks with two different IV the two resulting ciphertext sequences look completely unrelated for an attacker. IV is added to the first plaintext. We do not want to keep it secret because we use it as a nonce, so we replace it after a single use.

For the step $i$, the input of the cryptographic algorithm is the result between the XOR of the current plaintext block $B_i$ and the preceding encrypted block $y_{i-1}$; for each block, the same key $k$ is used. The plaintext $M$ is divided

into blocks $B_N||\dots B_2||B_1$, where each block has a required length given by the block cipher $E$ (e.g. 128 bits if $E =$ AES). If needed, there is a padding operation for the last block.
The formulas are the following:

$$y_1 = E_k(B_1 \oplus IV),$$

$$y_i = E_k(B_i \oplus y_{i-1}) \text{ if } i > 1.$$

When decrypting a ciphertext block $y_i$ in CBC mode, we have to reverse the two operations we have done while encrypting. First, we apply the decrypting function $D$, always with the same key $k$; then, we XOR the correct ciphertext block to undo the XOR done while encrypting. The formulas are

$$B_1 = D_k(y_1) \oplus IV,$$

$$B_i = D_k(y_i) \oplus y_{i-1} \text{ if } i > 1.$$

The main advantage of the CBC is the chain between all the blocks: in this way, changing a single bit from the IV or from the block $B_j$ changes $y_j$ and every encrypted block after $y_j$. Moreover, decryption is parallelizable.
Encryption itself is not sufficient: we also have to protect the integrity of the message. This can be achieved by MACs or digital signatures, which will be briefly discussed in the Sections 2.3 and 2.5.

## 2.2 Hash functions: SHA-256

**Definition 2.1.** *Let $\Sigma$ be an alphabet and let $\Sigma^*$ be the set of all words (of arbitrary length) obtained from $\Sigma$. A pre-Hash function is a function $h : \Sigma^* \to \Sigma^n$, with a fixed $n$.*

Usually, in most applications we take $\Sigma = \{0, 1\}$ and $n = 160, 256, 384$ or 512. $h(x)$ is called hash or digest; it follows from its definition that $h(x)$ can't be injective. Hash functions must have the "avalanche effect" property: changing a single bit in the input must change most of the bits of the output. A Hash function must satisfy some properties: it has to be one-way, collision resistant and second preimage resistant.

**Definition 2.2.** *Let $h(x)$ be a pre-Hash function. A couple of elements $(a, b) \in \Sigma^*$, with $a \neq b$ and with the property $h(a) = h(b)$ is called collision.*

**Definition 2.3.** *Let $h(x)$ be a pre-Hash function and let $z$ be a digest. $h(x)$ is said to be one-way if the computation of a number $x$ such that $h(x) = z$ is computationally infeasible .*

Given a digest, it should require a work equivalent to about $2^n$ hash computations to find any message that hashes to that value.

**Definition 2.4.** *Let $h(x)$ be a pre-Hash function and fix $a \in \Sigma^*$. $h(x)$ is second preimage resistant if the computation of an element $b \in \Sigma^*$ with $b \neq a$ and $h(a) = h(b)$ is computationally infeasible.*

**Definition 2.5.** *Let $h(x)$ be a pre-Hash function. $h(x)$ is collision resistant if the research of any collision $(a, b)$ is computationally infeasible.*

Finding any two messages which hash to the same value should require work equivalent to about $2^{\frac{n}{2}}$ hash computations. A known attack exploits the birthday paradox to generate collisions. Of course, collision resistant implies second preimage resistant.

**Definition 2.6.** *If a pre-Hash function is one-way, collision resistant and second preimage resistant, it is called Hash function.*

A mathematical proof of these properties is not trivial and usually they are just ensured from a lot of tries and experiments. Every Hash function available is guaranteed to be safe. The most famous and used are SHA-1, SHA-2 and SHA-3. WhatsApp utilizes a specific implementation of SHA-2, so we'll focus on this Hash Function for the rest of this Section.

SHA-2 (Secure Hash Algorithm 2) is a set of cryptographic Hash functions. The SHA-2 family consists of six Hash functions with digests (Hash values) which have 224, 256, 384 or 512 bits: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256. In particular, SHA-256 is a hash function computed with 32 bits words. It differs from SHA-512 only in the number of rounds. The other versions of SHA-2 are truncated versions of SHA-256 and SHA-512. SHA-2 was published in 2001 by the National Institute of Standards and Technology (NIST).

WhatsApp uses SHA-256 as the Hash function for HMAC (see the next Section) and SHA-512 as the Hash function for EdDSA signatures, which

will be discussed in Chapter 3. Since the two versions of SHA-2 are really similar, we just describe SHA-256.
First of all, the message $M$ which is going to be hashed is padded in such a way that its length is a multiple of 512 bits. The padding is done this way: if the length of the message $M$ is $l$ bits, append the bit 1 to the end of the message, and then append $k$ more zero bits, where $k$ is the smallest non-negative solution of the equation $l + 1 + k \equiv 448 \mod 512$. After this, append to the result the 64-bits block which is equal to the number $l$ written in binary. Then, $M$ is divided into blocks of length 512 bits, $M = B_N||\ldots B_2||B_1$. SHA-256 is built using the *Merkle-Damgård structure*: it uses a compression function $COM$ and an Initialization Vector IV. $COM$ is iterated $N$ times (the number of blocks of the plaintext). Let's call $H_j$ the hash obtained at iteration $j$. The general formulas are

$$H_1 = COM(IV, B_1),$$

$$H_j = COM(H_{j-1}, x_j) \text{ if } j > 1.$$

The output is $y = H_N$. $COM$ is derived from a block cipher $E$. This construction is called *Davies-Meyer* structure and works as follows: it uses the block $B_j$ as the key for the iteration $j$ of the compression function and $H_j$ as the block to encrypt. It begins with a fixed initial hash value $H_0$ (derived from the IV), and then it computes

$$H_j = E_{B_j}(H_{j-1}) \oplus H_{j-1} \ \forall j > 1.$$

The XOR is necessary because without it the function would be invertible and so breakable. $H_0$ is a sequence of eight, fixed, 32-bits words, which can be found in the document [16]. The SHA-256 compression function operates on a 512-bits message block and a 256-bits intermediate hash value. The 512 bits of the block $B_i$ are divided into sub-blocks $B_i^0, \ldots, B_i^{15}$, where each sub-block has now 32 bits. There are also 8 registers: let's call them $a, b, c, d, e, f, g, h$. In the rest of this Section, we'll use the following notation.

- $\oplus$: bitwise XOR;
- $\wedge$: bitwise AND;
- $\vee$: bitwise OR;

- $\bar{x}$: complement of a bit $x$;
- $+$: addition $\mod 2^{32}$;
- $>>>_n$: right rotation of $n$ bits;
- $R_n$: right shift of $n$ bits.

Then, the hash computation proceeds as follows:.

```
for $i = 1 : N$ {
  • Initialize the registers $a, b, c, d, e, f, g, h$ with the hash value $H_{i-1}$, divided into sub-blocks of 32 bits each. We denote these sub-blocks with $H^a_{i-1}, H^b_{i-1}, H^c_{i-1}, H^d_{i-1}, H^e_{i-1}, H^f_{i-1}, H^g_{i-1}, H^h_{i-1}$.
  • Apply the SHA-256 function to update the registers: it works as following.
    for $j = 0 : 63${
      – Compute the functions $Ch(e, f, g), Maj(a, b, c), \Sigma_0(a), \Sigma_1(e)$, and $W_j$, where:
          $Ch(x, y, z) = (x \wedge y) \oplus (\bar{x} \wedge z);$
          $Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z);$
          $\Sigma_0(x) = (>>>_2 x) \oplus (>>>_{13} x) \oplus (>>>_{22} x);$
          $\Sigma_1(x) = (>>>_6 x) \oplus (>>>_{11} x) \oplus (>>>_{25} x);$
          $\sigma_0(x) = (>>>_7 x) \oplus (>>>_{18} x) \oplus (R_3 x);$
          $\sigma_1(x) = (>>>_{17} x) \oplus (>>>_{19} x) \oplus (R_{10} x);$
          $W_j = B^i_j \text{ for } j = 0, \ldots, 15;$
          $W_j = \sigma_1(W_{j-2}) + W_{j-7} + \sigma_0(W_{j-15}) + W_{j-16} \text{ if } j \geq 16.$
      – $T_1 = h + \Sigma_1(e) + Ch(e, f, g) + K_j + W_j$, where $K_j$ is a constant word which changes iteration per iteration (check the document [16] for their definition);
      – $T2 = \Sigma_0(a) + Maj(a, b, c);$
      – $h = g;$
      – $g = f;$
```

- $f = e$;
- $e = d + T1$;
- $d = c$;
- $c = b$;
- $b = a$;
- $a = T1 + T2$.

}

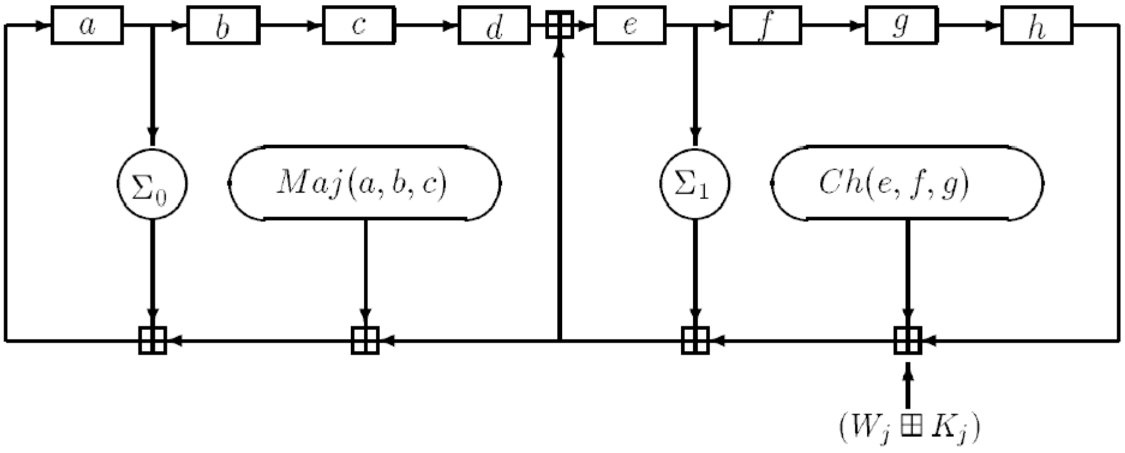


Figure 2.3: $j^{th}$ internal step of the SHA-256 Compression function

- Compute the $i^{th}$ intermediate hash value $H_i$: $H_i^a = a + H_{i-1}^a$, $H_i^b = b + H_{i-1}^b$ and so on for every other register;

}

$H_N = H_N^a||H_N^b||H_N^c||H_N^d||H_N^e||H_N^f||H_N^g||H_N^h$ is the hash of $M$.

Currently, the best public attacks break preimage resistance for 52 out of 64 rounds of SHA-256, and collision resistance for 46 out of 64 rounds of SHA-256. SHA-2 in general is weak to the *Length extension attack*, a kind of attack where it's possible to compute $H(m_1||m_2)$ only knowing $H(m_1)$ and the length of $m_1$ for an attacker-controlled message $m_2$. This is a flaw of the Merkle-Damgård construction. The use of a Message Authentication Code is necessary: MAC is not vulnerable to this attack, because it doesn't use this structure. The newest Hash function SHA-3 is not susceptible to this attack.

## 2.3 Message Authentication Code: MAC

A MAC consists of key generation, signing and verifying algorithms. MACs share integrity and authentication with digital signatures, but they don't give the non-repudiation property. A MAC takes a secret key and a message as input, and calculates a MAC output (a tag). The receiver can do the same operation, computing again the tag from the plaintext and the key. The operation has success if they get the same tag. Users have to share the secret key like they do when dealing with symmetric cryptography. They need to use a secure channel or they can exploit some properties of asymmetric cryptography to get a shared key, which must be pseudorandom (e.g., it can be generated by a PRNG). We'll see some examples in Chapter 3. MACs must not allow the computation of a correct tag for a new message for an user who doesn't know the key. A standard way to get this condition is utilizing a Hash function. In fact, the most used version of MAC is the Keyed-Hash Message Authentication Code (HMAC). HMAC can be implemented with every existing Hash function. WhatsApp, for example, uses HMAC with SHA-256. Let's see how HMAC works in this case. Let $H(x)$ be SHA-256, so it has as input sequences (blocks) of 512 bits and as output a digest of 256 bits. Let $K$ be the secret key: the best choice would be a key $K$ with length 256 bits, because a smaller key would be padded adding zeros and a bigger key would be hashed to this length. Define two fixed vectors,

$$\text{Ipad} = 00110110$$

and

$$\text{Opad} = 01011100,$$

where each string is repeated $\frac{256}{8} = 32$ times. Finally, let $M$ be the message. Then,

$$HMAC_K(M) = H((K' \oplus Opad)||H((K' \oplus Ipad)||M)),$$

where $K'$ is the key eventually padded or hashed if it is smaller or bigger than 256 bits.

This definition for the computation of the tag can look complicated, but it assures security against the length extension attack.

### 2.3.1 HMAC-based Key Derivation Function

A Key Derivation Function (KDF) is a basic and essential component of cryptosystems. Its goal is to take some source of initial keying material and derive from it one or more cryptographically strong secret keys. Usually, it takes weak key material (an example can be some Diffie-Hellman exchanged shared secrets which are not uniformly distributed) and a pseudo random KDF key, to turn it into a stronger key. This function consists of two steps: first, it takes the potentially weak source material and extracts from it a fixed-length pseudorandom key $k$; then, it can expand the key $k$ into several additional pseudorandom cryptographic keys.
The goal of the first stage is to "concentrate" the possibly dispersed entropy of the input keying material into a short, but cryptographically strong, pseudorandom key: if the input of the algorithm is an already secure key, the first step can be omitted and just the second step is performed. The goal of the second stage is instead the expansion of the pseudorandom key to the desired length, which depends on the cryptographic application. Let's now describe what happens in each stage.

- Step 1: Extraction.
  We need a Hash function (e.g. SHA-256): denote $l$ the length of the output of the Hash function in octets (for SHA-256, $l = 32$).
  The inputs for the extract phase are:

  - An optional *salt* value (i.e. a non secret random value). When a salt value isn't provided, a default value is used, precisely a byte sequence of zeros. The presence of a salt makes sure that the derived keys are different if the input keying material repeats;
  - IKM: an input key material. IKM is used as the HMAC input, not as the HMAC key.

  The HMAC function is used to derive the new key: the output is

$$PRK = HMAC_{salt}(IKM).$$

- Step 2: Expansion.
  We need again a Hash function. The inputs of the second phase are the following:

– PRK, a pseudorandom key of at least $l$ octets. Usually, it is the output of the first phase;
– *info*, an optional context and application specific information. It can be a string with $l = 0$. Its main objective is to bind the derived key material to application-specific information. For example, info may contain a protocol number or an ID. It should be independent from IKM;
– $L$, the length required for the output of this phase. It must satisfy the relation $L \leq 255l$.

Then, the output $OKM$ is computed in this way:

$$N = \left\lceil \frac{L}{l} \right\rceil,$$

$$T = T(1)||T(2)||\dots||T(N),$$

where

$$T(0) = \text{empty string},$$

$$T(i) = HMAC_{PRK}(T(i-1)||\mathit{info}||C_i);$$

$C_i$ is a constant which changes at every iteration and it is a byte (there is a maximum of 256 different constants).

Finally,

$$OKM = \text{first } \ L \ \text{ bytes of } T.$$

## 2.4 Authenticated-Encryption with Associated-Data

We have already discussed why confidentiality is not enough. The attacker might use active techniques, and the receiver might want to ensure the data have really been constructed by a sender with the right key. In the past, the common scheme was simply the composition of an encryption scheme and a MAC. More recently, the idea of providing both security services using a single cryptoalgorithm has been preferred: its name is Authenticated-Encryption with Associated-Data (AEAD). It provides confidentiality, integrity and data authenticity simultaneously. In fact, AEAD can authenticate a part of its input (e.g. a header), which is called Associated Data

($AD$). Associated Data are usually not encrypted.
Both cipher and MAC are replaced by an AEAD algorithm: it has more efficiency compared to the generic composition of conventional mechanisms and it is faster. Moreover, classic cipher and MAC schemes also have a significant shortcoming: an inability to efficiently authenticate a string of associated-data bound to the ciphertext. Instead, AEAD performs this operation without any problem. Several crypto algorithms that implement AEAD algorithms have been defined, including block cipher modes of operation and dedicated algorithms.
An AEAD algorithm has two operations, authenticated encryption and authenticated decryption. Most known ciphers can be used for these two computations, but the classic choice is AES with some mode. Authenticated encryption has 4 inputs, each of which is an octet string:

- a secret key $K$, which must be generated in a way that is uniformly random or pseudorandom. The length of $K$ must be fixed and it is between 1 and 255 octets;

- a nonce $N$, which changes at every iteration of the algorithm. For any particular value of the key, each nonce provided to distinct calls of the algorithm must be distinct, or each nonce must be zero-length;

- a plaintext $M$. The number of octets in $M$ may be zero;

- the associated data $AD$, which contain the data to be authenticated, but not encrypted. The number of octets in $AD$ may be zero;

The output is a ciphertext $C$, which is at least as long as the plaintext, or an indication that the requested encryption operation could not be performed. The number of octets in $C$ may be zero. This algorithm provides confidentiality and message authentication. If the length of $M$ is zero, the algorithm works like a MAC on $AD$. $AD$ is used to protect informations that need to be authenticated, but don't need to be kept confidential.

Similarly, the authenticated decryption operation has four inputs: $K$, $N$, $AD$, and $C$, as defined above. It has only a single output, either a plaintext $M$ or a special symbol FAIL that indicates that the inputs are not authentic.

Usually, AEAD is used like an encrypt-then-MAC scheme. In this scheme, the steps are the following: encrypt the cleartext, then compute the MAC on the ciphertext, and append it to the ciphertext. The keys for the two steps can be the same (even though it's not recommended) or different: in this case, the first $b$ bits of the key $K$ are used as the encryption key, and the last $b$ bits of $K$ are used as the authentication key.

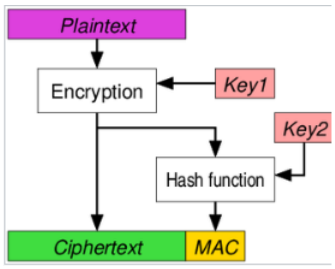


Figure 2.4: Encrypt-then-MAC scheme

## 2.5 Digital Signatures

When dealing with asymmetric cryptography, every user has a public key and a private key. If Alice ($A$) wants to send a message to Bob ($B$), she takes the public key of $B$ and encrypts her message. $B$ receives the message and is able to decrypt it thanks to her private key. $B$ doesn't know who sent him the message though. Digital signatures fix this problem, giving the recipient a way to check who sent the message. Thanks to signatures, we gain the property of non-repudation, i.e. the sender of a message can't deny its submission. Moreover, like MACs, a signature gives us the two properties of integrity and authentication. A general scheme for a digital signature is the following. Let $A$, $B$ be two users who want to exchange a message. Let $E_{k_a}, E_{k_b}$ be the two encrypting functions (they are public because they require the public keys), with inverse $D_{k_a}, D_{k_b}$ (they are private because they require the private keys), and let $h()$ be a Hash function. Let $s_A$ be a text containing some specific information about $A$, like her name, a progressive number or an ID. If A is the sender, the receiver B wants to be sure about who is sending him a message. The steps are the following:

- A sends to B $E_{k_b}(M)$ and the couple $(E_{k_b}(D_{k_a}(s_A||h(M))), s_A)$ where $h(M)$ is the digest of a plaintext $M$. The usage of $h()$ is important because it guarantees that this signature is viable only for this specific $M$;

- B obtains $M$ from $E_{k_b}(M)$ using his decrypting function $D_{k_b}$ and computes $h(M)$. Then, he applies to the signature he received the function $E_{k_a}D_{k_b}$ and subtracts $h(M)$: if the result he gets is equal to $s_A$, he is sure about the sender.

We'll see in the next Chapter a concrete example of digital signature on elliptic curves.

# Chapter 3

# Elliptic Curves

To understand how WhatsApp generates keys for its users, we need to introduce some theory about elliptic curves and their use in cryptography. This Chapter will face this argument, without worrying too much about mathematical details - the focus will be on the cryptographic applications. Elliptic Curve Cryptography (ECC) has been around since the mid-1980s. ECC provides the same level of security of other systems (such as Discrete Logarithm systems, DL for short) with smaller keys (approximately 160–256 bits vs. 1024–3072 bits). ECC is based on the generalized DL problem, and thus DL-protocols (such as the Diffie–Hellman key exchange) can also be realized using elliptic curves. In many cases, ECC needs less computations and uses shorter signatures and keys over the Discrete Logarithm scheme. An elliptic curve is a special type of polynomial equation. For cryptographic use, we need to consider the curve over a finite field. The most popular choice is the prime field $GF(q)$, where all arithmetic is performed modulo a prime $q$. The definition of an elliptic curve requires the curve to be nonsingular. Geometrically speaking, this means that the plot has no self-intersections or vertices, which is achieved if the discriminant of the curve is nonzero. We are not going to focus too much on this condition, since it's always possible to get a nonsingular curve without trouble. Section 3.1 will recap the most common and widely used type of elliptic curve, the curve in Short Weierstrass form. Sections 3.2 and 3.3 will introduce the recent Edwards curves, while Section 3.4 describes a last kind of elliptic curve, the Montgomery curve, and the important Montgomery Ladder algorithm. The Section 3.5 introduces Curve25519, which is the elliptic curve chosen from WhatsApp for the generation of keys. Finally, Section 3.6 describes two variants of a digital signature scheme defined on Edwards curves.

## 3.1 Short Weierstrass form curves

The most straightforward kind of elliptic curve can always be written in *Short Weierstrass form*, i.e. the curve has the equation $y^2 = x^3 + ax + b$ .

**Definition 3.1.1.** The elliptic curve over $GF(q)$, $q > 3$, is the set of all pairs $(x, y) \in GF(q)$ which fulfill

$$y^2 \equiv x^3 + ax + b \mod q \tag{3.1.1}$$

together with an imaginary point at infinity $\mathcal{O}$, where $a, b \in GF(q)$ and with the nonsingularity condition $4a^3 + 27b^2 \neq 0 \mod q$.

For cryptographic uses, we are interested in studying the curve over a finite field. However, nothing prevents us from taking an elliptic curve equation and plotting it over the set of real numbers.

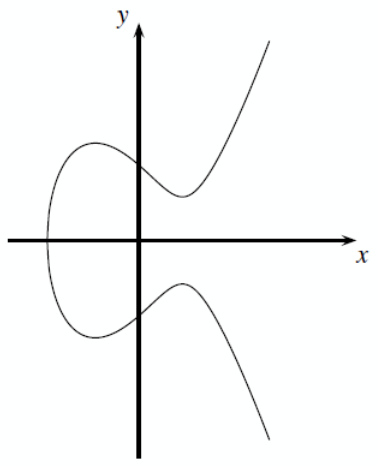


Figure 3.1: Plot of the Elliptic Curve $y^2 = x^3 - 3x + 3$ over $\mathbb{R}$

The elliptic curve is symmetric with respect to the $x$-axis (it follows from its definition). As already stated before, we need a group, i.e. we need some elements and an operation between these elements. Getting the elements of the group on an elliptic curve is easy: we can take the points (with integer coordinates) which fullfill the equation (3.1.1). The difficult part is finding an operation between these points. Let's denote this operation with "$\oplus$". If we know a few integer points on the elliptic curve $E$, we can generate new integer points on the curve by drawing a line passing through two distinct points $P$ and $Q$ or a tangent line passing for a point $P$. The new point of the curve $E$ where that line intersects the elliptic curve will be another integer point. This is known as the *Chord-Tangent Method.* If $P$, $Q$ are two points of $E$, let's call $S = (x, y)$ the point obtained using the Chord-Tangent method. Then, we define the operation on the elliptic curve as

$$R = P \oplus Q := (x, -y).$$

The same operation holds if we are considering a tangent line: just compute the point $R = P \oplus P$. Let's now get some formulas to calculate the new point $R$ starting from two other points $P$, $Q$ (or just a single point $P$). We need to distinguish between the two cases. Let's call $\lambda$ the slope of the line.

- Case 1: $R = P \oplus Q$ $(P = (x_P, y_P) \neq Q = (x_Q, y_Q))$ (point *addition*). We can compute the coordinates of $R$ with the formulas

$$x_R = \lambda^2 - x_P - x_Q \mod q \tag{3.1.2}$$
$$y_R = \lambda(x_P - x_R) - y_P \mod q \tag{3.1.3}$$
$$\lambda = \frac{y_Q - y_P}{x_Q - x_P} \mod q \tag{3.1.4}$$

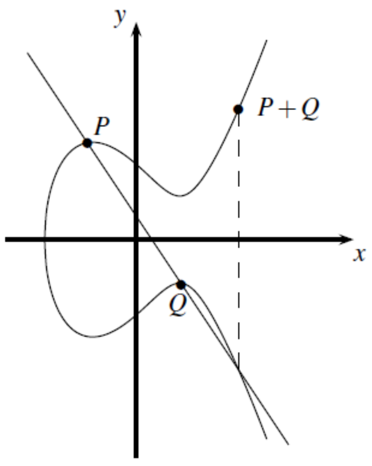


Figure 3.2: Point Addition

- Case 2: $R = P \oplus P = 2P$ $(P = Q = (x_P, y_P))$ (point *doubling*). We can compute the coordinates of $R$ with the formulas

(3.1.5) $$x_R = \lambda^2 - 2x_P \mod q$$

(3.1.6) $$y_R = \lambda(x_P - x_R) - y_P \mod q$$

(3.1.7) $$\lambda = \frac{3x_P^2 + a}{2y_P} \mod q$$

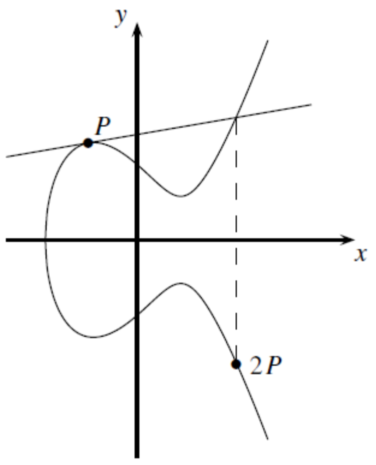


Figure 3.3: Point Doubling

The neutral element of the group is the "point at infinity" $\mathcal{O}$. It does not belong to the curve: we must use projective coordinates to fully describe this point. We can add a coordinate $z$ to get rid of $\mathcal{O}$. Without too much detail, we can think the curve with coordinates $(X : Y : Z)$, where this notation denotes the equivalence class of $(x, y, z)$. With these new coordinates, the equation of an elliptic curve in Short Weierstrass form is

$$Y^2Z = X^3 + aXZ^2 + bZ^3.$$

The classic form is just the curve $(X : Y : 1)$. We know that $\mathcal{O}$ does not belong to this plane, so $Z \neq 1$. We can take $Z = 0$ to get

$$0 = X^3 \Rightarrow X = 0.$$

The point $\mathcal{O}$ corresponds to the equivalence class $(0 : 1 : 0)$.
The inverse of a point $P$ is defined as $-P = (x, -y)$. In fact, it's easy to check that $P \oplus (-P) = \mathcal{O}$.
The associative property follows from some well-known results of algebraic

geometry: a proof can be found in [2].

We can reassume all these observations with a general Theorem.

**Theorem 3.1.2.** *Let $E$ be a rational elliptic curve. Denote $E(\mathbb{Q})$ as the set of all rational projective points on $E$. Then $E(\mathbb{Q})$ is an abelian group under $\oplus$.*

We can now also motivate the following result.

**Theorem 3.1.3.** *The points on an elliptic curve together with $\mathcal{O}$ have cyclic subgroups. Under certain conditions all points on an elliptic curve form a cyclic group.*

This Theorem is extremely useful because we have a good understanding of the properties of cyclic groups. For this reason, we know how to build cryptosystems from cyclic groups.

**Definition 3.1.4.** The number of points in a group is called *order* of the group. The order of a point $P$ is the smallest positive integer $n$ such that $nP = \mathcal{O}$.

The order of $P$ is linked to the order of the elliptic curve by the Lagrange's Theorem, which states that the order of a subgroup is a divisor of the order of the group. In other words, if an elliptic curve contains $N$ points and one of its subgroups contains $n$ points, then $n$ is a divisor of $N$. We call *cofactor* the ratio $c = \frac{N}{n}$. The number of points $N$ can be computed efficiently, for example, with the Schoof's algorithm, which has $O(log^9(q))$ complexity. The details can be found in [5].
Instead, an estimate of the number of points $N$ of an elliptic curve is given from this Theorem.

**Theorem 3.1.5.** *(Hasse)*
*Let $N$ be the number of points of an elliptic curve $E(GF(q))$, where $q$ is the power of a prime. Then,*

$$|N - (q + 1)| \leq 2\sqrt{q}.$$

This Theorem states that the number of points is roughly in the range of the prime $q$. For instance, if we need an elliptic curve with $2^{160}$ elements, we have to use a prime of length of about 160 bits.

The second step is the description of the discrete logarithm problem on an elliptic curve. Let's consider a primitive element $P$ and another element $T$. The Elliptic Curve Discrete Logarithm Problem (ECDLP) consists in finding a number $d$ such that $dP = T$, where with $dP$ we denote the computation $P \oplus \cdots \oplus P$ $d$ times. In cryptosystems, $d$ is the private key which is an integer, while the public key $T$ is a point on the curve with coordinates $T = (x_T, y_T)$. If the elliptic curve is chosen with care, the best known attacks against the ECDLP are considerably weaker than the best algorithms for solving the DL problem modulo $q$. It is important to remember that this security is only achieved if cryptographically strong elliptic curves are used. There are several families of curves that possess cryptographic weaknesses, like supersingular curves. A core requirement is that the cyclic group (or subgroup) formed by the curve points has prime order. There are other properties to be respected and assuring all of them is not trivial: usually, people use standardized curves.

**Example 3.1.** *(ECDSA)*
*Before moving on with the description of other kinds of elliptic curves used in cryptography, let's describe the Elliptic Curve Digital Signature Algorithm (ECDSA). A variant of this digital signature is used by WhatsApp. Fix an elliptic curve in Short Weierstrass form $E$ in the field $GF(q)$ and a point $G$ on the curve. Let $N$ be the number of points of the curve (it can be found with Schoof's algorithm) and let $H$ be a Hash function. Every user chooses an integer $d \in GF(q)$ as her private key, computes $P = dG$ and uses $P$ as her public key.*
*ECDSA signing algorithm works as follows:*

- *the sender $A$ computes the digest $h = H(M)$ of the plaintext $M$;*
- *$A$ chooses $k \in \mathbb{Z}_N$ randomly (i.e. every value has the same probability, the distribution is uniform);*
- *$A$ computes the point of the curve $kG = (x, y)$;*
- *$A$ fixes $r \equiv x \mod N$ and computes $s = (h + rd)k^{-1} \mod N$;*
- *The signature is the couple $(r, s)$.*

*It guarantees integrity, authentication and non-repudiation, because only $A$ can know her private key $d$, so only $A$ can compute $(r, s)$. Moreover, in the signature there is $h = H(M)$, so it can be used only for the specific message $M$, it can't be reused fraudulently for other messages. To verify the signature, the receiver $B$ must:*

- *compute* $w = s^{-1} \mod N = k(h + rd)^{-1} \mod N$*;*
- *compute* $u = wh$ *and* $v = wr$*;*
- *compute the point* $Q = uG + vP$*, where* $P = dG$ *is the public key of* $A$*;*
- *accept the signature only if the first coordinate of the point* $Q$*, let's say* $x$*, coincides with* $r \mod N$*.*

*This works because*

$$Q = uG + vP = whG + wr(dG) = w(h + rd)G = kG = (x, y).$$

## 3.2 Edwards curves

Edwards curves are a family of elliptic curves studied by Harold Edwards in 2007. Let $GF(q)$ be a field with char$(GF(q)) \neq 2$. The starting point to face this family of curves is the equation of the unitary circumference $x^2 + y^2 = 1$. We can take two points of the circumference and "add" them (denote again this operation with the symbol $\oplus$), getting a new point of the circumference, simply using goniometric formulas.

In fact, let's take two random points $P_1$ and $P_2$, and let's call $\alpha$ the angle between the $y$-axis and $P_1$, $\beta$ the angle between the $y$-axis and $P_2$. In this way, we can rename the coordinates of the points as

$$P_1 = (\sin(\alpha), \cos(\alpha)),$$

$$P_2 = (\sin(\beta), \cos(\beta)).$$

The addition of angles defines a commutative group law:

$$P_3 = P_1 \oplus P_2 = (\sin(\alpha + \beta), \cos(\alpha + \beta)),$$

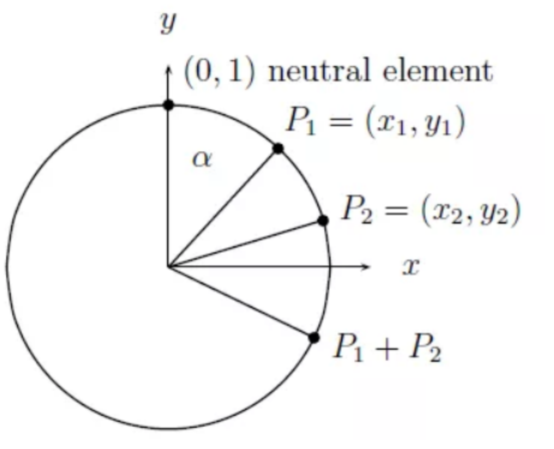


Figure 3.4: Sum of points on a circumference

where

$$\sin(\alpha+\beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)$$

and

$$\cos(\alpha+\beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta).$$

The neutral element is the point $(0, 1)$, and $P_3$ satisfies the unit circle equation. Sadly, it can be proved that a circumference is not an elliptic function.

Generalizing the example, we get the so-called *Edwards curves*.

**Definition 3.2.1.** An Edwards curve over $GF(q)$, with $\text{char}(GF(q)) \neq 2$ is a curve given by the equation

$$x^2 + y^2 = 1 + dx^2y^2.$$

where $d \in GF(q)$ with $d \neq 0, 1$.

If $d$ becomes smaller and smaller, the curve looks more and more like a "starfish". We have to define the elements of the group and the operation between these elements. Similarly to the Short Weierstrass curves case, the research of the elements is easy: we just take the points of the curve with integer coordinates. Surprisingly, the operation in this kind of elliptic curve is really simple and doesn't use the chord and tangent method. The procedure remembers the unit circle's addition law, and an idea of its construction can be found in [34]. The neutral element is also the same, i.e. the point $(0, 1)$.

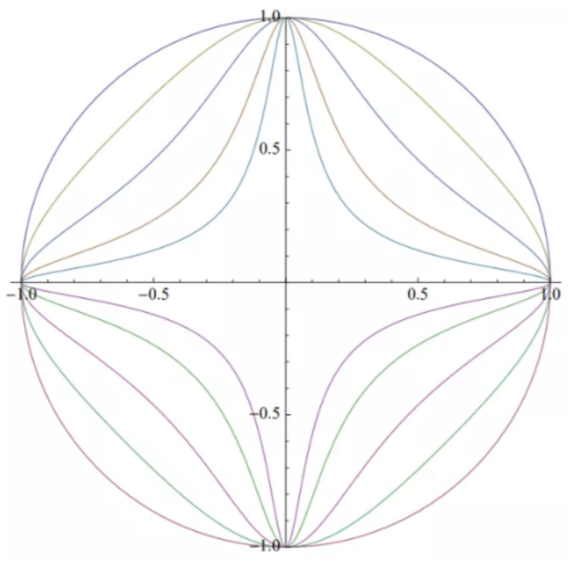


Figure 3.5: Edward curves for $d = -2, -10, -50, -200$.

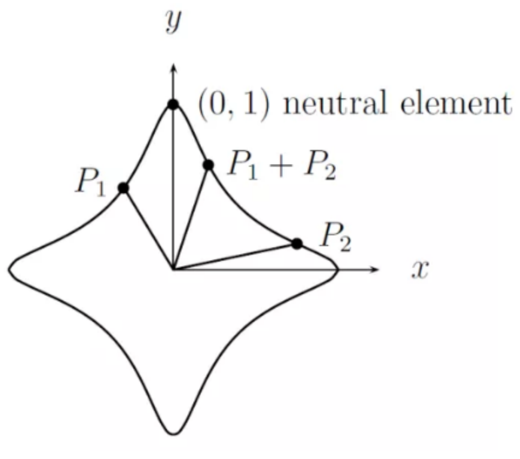


Figure 3.6: Operation on an Edward curve

The formulas for the addition are the following: if $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, then

$$P_3 = P_1 \oplus P_2 = \left( \frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2} \mod q, \frac{y_1y_2 - x_1x_2}{1 - dx_1x_2y_1y_2} \mod q \right).$$

Doubling can also be performed with the same formula. It's also easy to prove that, if $P = (x, y)$, then $-P = (-x, y)$.
Bernstein and Lange generalized the addition law to the curves with equation $x^2 + y^2 = c^2(1 + dx^2y^2)$ $(c \neq 0)$. All curves in this form are isomorphic to the curves in the classical Edwards form. Let's now compute the order of some trivial points of the curve, the "angles" $(0, -1)$, $(1, 0)$ and $(-1, 0)$.

Using the doubling formula, we get

$$(0,-1) \oplus (0,-1) = (0,1),$$

$$2(1,0) = (1,0) \oplus (1,0) = (0,-1) \Rightarrow 4(1,0) = (0,1),$$

$$2(-1,0) = (-1,0) \oplus (1,0) = (0,-1) \Rightarrow 4(-1,0) = (0,1),$$

so the points have order 2, 4, 4 respectively. Trivially, these points belong to every Edwards curve: this characteristic will be important later.
How these curves relate with classical elliptic curves? Edwards showed that every elliptic curve over $GF(q)$ can be expressed in the form $x^2 + y^2 = 1 + dx^2y^2$ if $GF(q)$ is algebraically closed. However, over a finite field, only a small fraction of elliptic curves can be expressed in this form. To be more precise, we need to introduce the concept of *birational equivalence.*

**Definition 3.2.2.** Two curves $E_1$ and $E_2$ are *birationally equivalent* if there is a map $\phi : E_1 \to E_2$ between them which is defined at every point of $E_1$ except a small set of exceptions and there is an inverse map $\phi^{-1} : E_2 \to E_1$ which is defined at every point of $E_2$ except a small set of exceptions.

This definition is very close to that of an isomorphism, except for the fact that we allow some "exceptions", i.e. points where the map is not defined. To make all more concrete, we can think of an isomorphism as a tuple of polynomials:

$$\phi : E_1 \to E_2$$

$$(x,y) \mapsto (f(x,y), g(x,y)),$$

where $f, g$ are polynomials in $x, y$. The inverse is also defined in terms of polynomials.
A birational map can be seen as a tuple of fractions of polynomials, say

$$\phi : E_1 \to E_2$$
$$(x,y) \mapsto \left(\frac{f_1(x,y)}{f_2(x,y)}, \frac{g_1(x,y)}{g_2(x,y)}\right).$$

This is defined at all points $(x,y)$ except for the ones where $f_2(x,y) = 0$ or $g_2(x,y) = 0$. The inverse is also a fraction of polynomials, and can therefore be undefined at certain points.

What is true is that every elliptic curve in Short Weierstrass form $E$, defined on a finite field, with a point of order 4 (and with a single point of order 2) is birationally equivalent to an Edwards curve $E_d$. This condition is natural because we have already observed that every Edwards curve has always a point of order 4. There are no NIST curves at the moment with points of order 4, so to build the birational equivalence we must work on some extension field with a point of order 4.
In particular, this equivalence means that the output of the Edwards addition law corresponds to the output of the standard addition law on a birationally equivalent elliptic curve $E$. One can therefore perform group operations on $E$ (or on any other birationally equivalent elliptic curve) by performing the corresponding group operations on the Edwards curve. The group operations could encounter exceptional points where the Edwards addition law is not defined. It can be proved that, when $d$ is not a square, there are no exceptional points: the denominators in the Edwards addition law cannot be zero (see [8] for details about the proof). In other words, when $d$ is not a square, the Edwards addition law is complete: it is defined for all pairs of input points on the Edwards curve over $GF(q)$. If $q \equiv 3 \mod 4$, this pitfall can be avoided choosing a curve with order $4r$, where $r$ is a prime.

## 3.3 Twisted Edwards curves

We can generalize Edwards curves introducing a new parameter $a$. In fact, the existence of points of order 4 restricts the number of elliptic curves in Edwards form over $GF(q)$. We embed the set of Edwards curves in a larger set of elliptic curves of a similar shape by introducing *twisted Edwards curves*. First of all, we need the following definition.

**Definition 3.3.1.** Let $E$ be an elliptic curve over a field $GF(q)$. We call an associated *quadratic twist* another elliptic curve which is isomorphic to $E$ over an algebraic closure of $GF(q)$.

*Remark* 3.3.2. An algebraic closure of a field $K$ is an algebraic extension of $K$ that is algebraically closed, so it contains a root for every non-constant polynomial in $K[x]$, the ring of polynomials in the variable $x$ with coefficients in $K$.

Then, we can define the *twisted Edwards curves.*

**Definition 3.3.3.** Fix a field $GF(q)$ with $\text{char}(GF(q)) \neq 2$. Fix distinct nonzero elements $a, d \in GF(q)$. The twisted Edwards curve with coefficients $a$ and $d$ is the curve

$$E_{E,a,d} : ax^2 + y^2 = 1 + dx^2y^2.$$

An Edwards curve is a twisted Edwards curve with $a = 1$.
The twisted Edwards curve $E_{E,a,d} : ax^2+y^2 = 1+dx^2y^2$ is a quadratic twist of the Edwards curve $E_{E,1,\frac{d}{a}} : \bar{x}^2 + \bar{y}^2 = 1 + \frac{d}{a}\bar{x}^2\bar{y}^2$.

The group law for twisted Edwards curves is really similar to the one of Edwards curves. The formula is the following: if $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, then

$$P_3 = P_1 \oplus P_2 = \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2} \mod q, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2} \mod q\right).$$

The formula is the same for point doubling. These formulas are complete (i.e., have no exceptional cases) if $a$ is a square in $GF(q)$ and $d$ is a nonsquare in $GF(q)$. Again, if $q \equiv 3 \mod 4$, this can be avoided choosing a curve with order $4r$, $r$ prime. This generalization is actually useful for cryptographic purposes because there is often a computational advantage: less operations are required in average. Even when an elliptic curve can be expressed in Edwards form, expressing the same curve in twisted Edwards form often saves time in arithmetic. It can be proved (read [10] for details) that if $q \equiv 1$ mod 4, twisted Edwards curves cover considerably more elliptic curves than Edwards curves.

**Example 3.2.** *(Edwards448)*
*Edwards448 is one of the most used non-standardized curves, together with Curve25519, which will be discussed later. It is used by Signal, but not by WhatsApp. A reason is the fact that on this curve, while it offers more security (224 bits of security), computations are slower. The equation of the curve is*

$$x^2 + y^2 = 1 - 39081x^2y^2,$$

*defined over $GF(q)$ with $q = 2^{448} - 2^{224} - 1$. It can be proved that $q \equiv 3 \mod 4$. The constant $d = -39081$ was chosen as the smallest number (in absolute value) which guarantees the completeness of the law (i.e. the curve and its twist have order $4r$ and $4r'$ with $r, r'$ primes).*

## 3.4 Montgomery curves

We now introduce a last class of elliptic curves.

**Definition 3.4.1.** Fix a field $GF(q)$ with $\mathrm{char}(GF(q)) \neq 2$. Fix $A \in GF(q)$ with $A \neq \pm 2$ and $B \in GF(q)$, $B \neq 0$. The *Montgomery curve* with coefficients $A$ and $B$ is the curve

$$E_{M,A,B} = By^2 = x^3 + Ax^2 + x.$$

The parameter $A$ controls the geometry of the curve, while the parameter $B$ is called twisting factor. Since the value of $B$ is incidental for cryptography, we won't give too much importance to this parameter. $E_{M,A,B}$ is an elliptic curve, so there must be a group law between its points, with operation $\oplus$. In this case, it is defined in the following way: let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ be two points of the curve, and let $\lambda$ be the slope of the line passing for the two points. Then, $R = P \oplus Q = (x_R, y_R)$, with

$$x_R = B\lambda^2 - (x_P + x_Q) - A \mod q \tag{3.4.1}$$

$$y_R = \lambda(x_P - x_R) - y_P \mod q \tag{3.4.2}$$

and

$$\lambda = \frac{y_Q - y_P}{x_Q - x_P} \mod q$$

if $P \neq \pm Q$,

$$\lambda = \frac{3x_P^2 + 2Ax_P + 1}{2By_P} \mod q$$

otherwise. The neutral element is the point at infinity $\mathcal{O}$, while the inverse of the point $P = (x, y)$ is the point $-P = (x, -y)$.
For Montgomery curves, the following Theorem holds.

**Theorem 3.4.2.** *Fix a field $GF(q)$ with char$(GF(q)) \neq 2$. Then:*

- *Every twisted Edwards curve over $GF(q)$ is birationally equivalent over $GF(q)$ to a Montgomery curve.*
- *Conversely, every Montgomery curve over $GF(q)$ is birationally equivalent over $GF(q)$ to a twisted Edwards curve.*

It's important to observe that this is not true for curves in Short Weierstrass form: every Montgomery curve can be written in Short Weierstrass form (if $q$ is not a power of 3), while the converse is not true in general. From the Theorem, it follows that there is a quadratic twist of $E_{M,A,B}$ birationally equivalent to an Edwards curve. The transformation formulas from the twisted Edwards curve $ax^2 + y^2 = 1 + dx^2y^2$ to the Montgomery curve $Bv^2 = u^3 + Au^2 + u$ are

$$u = \frac{1+y}{1-y} \Rightarrow y = \frac{u-1}{u+1} \mod q$$
$$v = \frac{1+y}{x(1-y)} \Rightarrow x = \frac{u}{v} \mod q$$

Doing the calculations, we find $A = \frac{2(a+d)}{a-d}$ and $B = \frac{4}{a-d}$.
The map is defined for every point if $\infty$ is handled carefully as input and output. The critical points are $u = -1$ and $v = 0$.

**Example 3.3.** *(Curve25519)*
*Let's consider the Montgomery curve known as Curve25519. It will be fundamental in the generation of keys for the WhatsApp key exchange protocol. This curve offers 128 bits of security and it is defined as*

$$v^2 = u^3 + 486662u^2 + u,$$

*defined over $GF(q)$ with $q = 2^{255} - 19$.*

*Let's apply the transformation described above. we have $B = 1$ and $A = 486662$. Solving the system of equations, we get a and d:*

$$\begin{cases} \frac{4}{a-d} = 1 \\ \frac{2(a+d)}{a-d} = 486662 \end{cases}$$
$$\Rightarrow a = 486664 \quad d = 486660.$$

*The twisted Edwards curve is $486664x^2 + y^2 = 1 + 486660x^2y^2$. We know too that this curve is equivalent to the Edwards curve $E_d : x^2 + y^2 = 1 +$*

$\frac{121665}{121666}x^2y^2$. *Thanks to this relation, we have proved that Curve25519 has a point of order* 4. *It can also be shown that* $q \equiv 1 \mod 4$. *Doing a different trasformation, it can be shown that the twisted Edwards curve can be written in the form* $-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2$ *also.*

There are some situations where the twisting is not necessary. We already know that every elliptic curve in Short Weierstrass form $E$ with a point of order 4 is birationally equivalent to an Edwards curve $E_d$. The result is generalized in the following Theorem.

**Theorem 3.4.3.** *Fix a field* $GF(q)$ *with char(*$GF(q)$*)* $\neq 2$. *Let* $E$ *be an elliptic curve over* $GF(q)$. *The group* $E(GF(q))$ *has an element of order* 4 *if and only if* $E$ *is birationally equivalent over* $GF(q)$ *to an Edwards curve.*

This also means that the unique point of order 2 is unnecessary.

All variants of elliptic curves described above allow a key exchange protocol. Miller, and independently Koblitz, proposed an elliptic curve variant of the classic Diffie-Hellman method.

*Remark* 3.4.4. Elliptic curve Diffie-Hellman (ECDH) is a generalization of the Diffie-Hellman protocol on elliptic curves. Let's see how it works on an Elliptic curve written in Short Weierstrass form with coefficients in $\mathbb{Z}_p$. Let $P$ be the (public) base point of the elliptic curve. Alice and Bob choose their secret keys $k_A, k_B \in \mathbb{Z}_p$ and compute the public keys $P_A = k_AP$ and $P_B = k_BP$. They can now obtain a shared key: if Alice wants to write a message to Bob, she takes the value $P_B$ and computes $k_1 = k_AP_B = k_Ak_BP$. In the same way, Bob takes the value $P_A$ and computes $k_2 = k_BP_A = k_Bk_AP$. Since we are in an abelian group, $k_Ak_B = k_Bk_A$ and for this reason they obtain the same point of the curve. They can now use this key to exchange messages through a symmetric cipher like AES.

Miller suggested to exchange just the $x$-coordinate instead of $(x, y)$-coordinates: i.e., sending $\mathbf{x}(P)$ rather than an entire point $P$, where $\mathbf{x}(x, y) = x$. In this case, ECDH works as follows: let's assume we have a curve written in Short Weierstrass or Montgomery form, the same base point $P$ and the same private keys of the Remark. Alice computes $\mathbf{x}(k_AP)$, while Bob computes $\mathbf{x}(k_BP)$. Alice and Bob then both know a shared secret

$\mathbf{x}(k_Ak_BP) = \mathbf{x}(k_A(k_BP)) = \mathbf{x}(k_B(k_AP))$. The algorithm works as follows: suppose that A wants to send a message to B. Then:

- A gets the public key of B $\mathbf{x}(k_BP)$, substitutes the value in the equation of the curve and solves the equation, finding two possible values of $y$. The values are always 2 because $\mathbf{x}(k_BP)$ is the coordinate of a point of the curve and we are working in a field;
- A has now two possible points $\pm k_BP = (\mathbf{x}(k_BP), \pm y)$;
- A computes the two points $\pm k_Ak_BP$ with the Montgomery formulas of addition and doubling: the computation of the $y$-coordinate is necessary for this step;
- these two points have the same $x$-coordinate: A has now the shared key $\mathbf{x}(k_Ak_BP)$.

Observe that $\mathbf{x}(k_Ak_BP)$ is entirely determined by $k_A$ and $\mathbf{x}(k_BP)$. Indeed, the only possible ambiguity in recovering $k_BP$ from $\mathbf{x}(k_BP)$ is the possible distinction between $k_BP$ and $-k_BP$, and this distinction has no effect on $\mathbf{x}(k_Ak_BP)$: the $x$-coordinate is invariant under point negation, so $\mathbf{x}(k_A(-k_BP)) = \mathbf{x}(-k_Ak_BP) = \mathbf{x}(k_Ak_BP)$. The same argument applies if $x$-coordinates on Short Weierstrass or Montgomery curves are replaced by $y$-coordinates on (twisted) Edwards curves. The bottleneck here is elliptic-curve scalar multiplication: the operations could require too much time.
For Montgomery curves, A can use the more efficient *Montgomery Ladder* to compute $\mathbf{x}(k_Ak_BP)$ from $\mathbf{x}(k_BP)$ and $k_A$, using the doubling and differential-addition formulas. This approach is simpler and almost 3 times faster than the algorithm just described.
The structure of Montgomery curves is important for the speed: from the modern Edwards perspective, Montgomery takes advantage of having a point of order 4 on the curve or its twist.
First of all, let's see how to compute $\mathbf{x}(nP)$ knowing only $n$ and $\mathbf{x}(P)$. Let's start with the simplest case, $n = 2$. .

- **Doubling**

  The following Theorem holds.

  **Theorem 3.4.5.** *Fix a field $GF(q)$ with char($GF(q)$) $\neq 2$. Fix $A, B \in GF(q)$ with $B(A^2-4) \neq 0$. Define $M$ as the Montgomery curve $By^2 =$*

*$x^3 + Ax^2 + x$, and define $\boldsymbol{x}$ as follows:*

$$\boldsymbol{x} : M(GF(q)) \to GF(q) \cup \{\infty\}$$

$$\boldsymbol{x}(x, y) = x, \quad \boldsymbol{x}(\mathcal{O}) = \infty.$$

*Let $P = (x, y)$ be an element of $M(GF(q))$. If $\boldsymbol{x}(P) = \infty$, then $\boldsymbol{x}(2P) = \infty$. If $\boldsymbol{x}(P) \neq \infty$ and $\boldsymbol{x}(P)^3 + A\boldsymbol{x}(P)^2 + \boldsymbol{x}(P) = 0$ then $\boldsymbol{x}(2P) = \infty$. If $\boldsymbol{x}(P) \neq \infty$ and $\boldsymbol{x}(P)^3 + A\boldsymbol{x}(P)^2 + \boldsymbol{x}(P) \neq 0$, then*

$$\boldsymbol{x}(2P) = \frac{(\boldsymbol{x}(P)^2 - 1)^2}{4(\boldsymbol{x}(P)^3 + A\boldsymbol{x}(P)^2 + \boldsymbol{x}(P))}.$$

*Proof.* If $\mathbf{x}(P) = \infty \Rightarrow P = \mathcal{O} \Rightarrow 2P = \mathcal{O} \Rightarrow \mathbf{x}(2P) = \infty$. Assume now that $\mathbf{x}(P) \neq \infty$. Then, $P = (x, y)$ for some $x, y \in GF(q)$ and $x, y$ satisfy the equation of $M$. By definition, $\mathbf{x}(P) = \mathbf{x}(x, y) = x$. If we assume $x^3 + Ax^2 + x = 0 \Rightarrow y = 0$. Then,

$$2P = (x, 0) + (x, 0) = (x, 0) - (x, 0) = \mathcal{O} \Rightarrow \mathbf{x}(2P) = \infty,$$

where the second equality follows from the fact that if $P = (x, y)$ then $-P = (x, -y)$ for a curve in Montgomery form. Assume now $x^3 + Ax^2 + x \neq 0 \Rightarrow y \neq 0$. We can use the formulas (3.4.1 - 3.4.2) with $\lambda = \frac{3x^2+2Ax+1}{2By} \mod q$ (because we are doubling). Consequently,

$$\mathbf{x}(2P) = B\lambda^2 - 2x - A = B\left(\frac{3x^2 + 2Ax + 1}{2By}\right)^2 - A - 2x$$

$$= \frac{(3x^2 + 2Ax + 1)^2}{4By^2} - A - 2x$$

$$= \frac{(3x^2 + 2Ax + 1)^2}{4(x^3 + Ax^2 + x)} - A - 2x = \cdots = \frac{x^4 - 2x^2 + 1}{4(x^3 + Ax^2 + x)} = \frac{(x^2 - 1)^2}{4(x^3 + Ax^2 + x)}$$

where in the third passage we have used the equation of the Montgomery curve $M$ and then we have just done basic calculations to get the thesis. $\square$

Divisions are slow. To avoid divisions, the Montgomery Ladder represents $x$-coordinates in a slightly different way. This requires extra-multiplications, but there is still a computational gain. Using projective coordinates, the Theorem can be adjusted: it is possible to prove that these formulas are complete. This is over the scope of the thesis. The interested reader can check, for example, [11].

- **Differential addition**

  We would like to find a formula to compute, for example, $\mathbf{x}(P_2 + P_3)$ starting from $\mathbf{x}(P_2)$ and $\mathbf{x}(P_3)$. Sadly, this is not possible because, if we have only these informations, we can have multiple possible values for $\mathbf{x}(P_2+P_3)$. However, the computation becomes certain if we know $\mathbf{x}(P_3-P_2)$ too. These formulas can, for example, produce $\mathbf{x}(3P)$ given $\mathbf{x}(2P)$ and $\mathbf{x}(P)$, or produce $\mathbf{x}(7P)$ given $\mathbf{x}(4P)$, $\mathbf{x}(3P)$, $\mathbf{x}(P)$.

**Theorem 3.4.6.** *Fix a field $GF(q)$ with char($GF(q)$) $\neq 2$. Fix $A, B \in GF(q)$ with $B(A^2-4) \neq 0$. Define $M$ as the Montgomery curve $By^2 = x^3 + Ax^2 + x$, and define $\boldsymbol{x}$ as follows:*

$$\boldsymbol{x} : M(GF(q)) \to GF(q) \cup \{\infty\}$$

$$\boldsymbol{x}(x, y) = x, \quad \boldsymbol{x}(\mathcal{O}) = \infty.$$

*Let $P_2$, $P_3$ be elements of $M(GF(q))$ with $P_3 \neq \mathcal{O}$, $P_2 \neq \mathcal{O}$, $P_2 \neq P_3$ and $P_3 \neq -P_2$. Then, $\boldsymbol{x}(P_3) \neq \boldsymbol{x}(P_2)$ and*

$$\boldsymbol{x}(P_3 + P_2)\boldsymbol{x}(P_3 - P_2) = \frac{(\boldsymbol{x}(P_3)\boldsymbol{x}(P_2) - 1)^2}{(\boldsymbol{x}(P_3) - \boldsymbol{x}(P_2))^2}.$$

We omit the proof, which uses the same ideas of the proof of Theorem 3.4.5. Even in this case, formulas can be optimized to remove the division. Using projective coordinates, it is possible to prove that these formulas are quasi-complete: in fact, there are only two "problematic" points, $(0, 0)$ and $\mathcal{O}$. These points always produce the same output and can be handled changing slightly the definition of $\mathbf{x}$ in the following way:

$$\mathbf{x}_0 : M(GF(q)) \to GF(q) \cup \{\infty\}$$

$$\mathbf{x}_0(x, y) = x, \quad \mathbf{x}_0(\mathcal{O}) = 0.$$

With this function, the formulas are complete. A proof for this fact can be found, again, in [11].

### 3.4.1 The Montgomery Ladder

This Section combines the Montgomery's doubling formula with the Montgomery's differential-addition formula to obtain the Montgomery Ladder.

The algorithm has full generality and applies to any abelian group. Originally, the Montgomery Ladder was proposed to speed up scalar multiplication in the context of elliptic curves. Let's start describing the algorithm in general. We want to compute $y = g^k$ in an abelian group $(G, *)$, with inputs $g$ and $k$. Let's write the binary expansion of the exponent $k$:

$$k = \sum_{i=0}^{t-1} k_i 2^i, \quad k_i \in \{0, 1\} \;\; \forall i.$$

The Montgomery Ladder relies on the following observation. Define

$$L_j := \sum_{i=j}^{t-1} k_i 2^{i-j},$$

$$H_j := L_j + 1.$$

Observe that $L_0 = k$. Then, it's easy to prove that

$$(L_j, H_j) = \begin{cases} (2L_{j+1}, 2L_{j+1} + 1) & \text{if } k_j = 0 \\ (2L_{j+1} + 1, 2L_{j+1} + 2) & \text{if } k_j = 1. \end{cases}$$

$$= \begin{cases} (2L_{j+1}, L_{j+1} + H_{j+1}) & \text{if } k_j = 0 \\ (L_{j+1} + H_{j+1}, 2H_{j+1}) & \text{if } k_j = 1. \end{cases}$$

Suppose that at each iteration, a first register $R_0$ contains the value of $g^{L_j}$ and a second register $R_1$ contains the value of $g^{H_j}$ . The prior system implies that

$$(g^{L_j}, g^{H_j}) = \begin{cases} ((g^{L_{j+1}})^2, g^{L_{j+1}} g^{H_{j+1}}) & \text{if } k_j = 0 \\ (g^{L_{j+1}} g^{H_{j+1}}, (g^{H_{j+1}})^2) & \text{if } k_j = 1. \end{cases}$$

This algorithm for evaluating $y = g^k$ is called the Montgomery Ladder. The starting point is always $L_j = 0$, $H_j = 1$: for this reason, we have $R_0 = 1$ and $R_1 = g$ as the start of the algorithm. Algorithm' steps are reassumed in Figure 3.7. The output of the algorithm is of course $g^k = g^{L_0}$.
From a computational perspective, the Montgomery Ladder, in its basic version, appears inferior to other classic binary algorithms as it requires $2t$ multiplications instead of $1.5t$ multiplications, on average.
The key property is that the relation $\frac{R_1}{R_0}$ remains constant $(= g)$ during the iterations of the algorithm. This fact can be applied on Montgomery curves. Let $M$ be a Montgomery curve. Let $R_0$ and $R_1$ be two points of $M$ defined

```
Input: $g, k = (k_{t-1}, \ldots, k_0)_2$
Output: $y = g^k$
$R_0 \leftarrow 1$; $R_1 \leftarrow g$
for $j = t-1$ downto 0 do
  if $(k_j = 0)$ then
    $R_1 \leftarrow R_0 R_1$; $R_0 \leftarrow (R_0)^2$
  else [if $(k_j = 1)$]
    $R_0 \leftarrow R_0 R_1$; $R_1 \leftarrow (R_1)^2$
return $R_0$
```

Figure 3.7: Montgomery Ladder

over $GF(q)$. If the difference $G = R_1 - R_0$ is known then $\mathbf{x}(Y) = \mathbf{x}(kG)$ can be computed from $\mathbf{x}(R_0)$, $\mathbf{x}(R_1)$ and $\mathbf{x}(G)$. (In the Figure 3.8, $+$ is

```
Input: $\boldsymbol{G}, k = (1, k_{t-2}, \ldots, k_0)_2$
Output: $\boldsymbol{Y} = k\boldsymbol{G}$
$\boldsymbol{R_0} \leftarrow \boldsymbol{G}$; $\boldsymbol{R_1} \leftarrow 2\boldsymbol{G}$
for $j = t-2$ downto 0 do
  if $(k_j = 0)$ then
    $\boldsymbol{R_1} \leftarrow \boldsymbol{R_0} + \boldsymbol{R_1}$; $\boldsymbol{R_0} \leftarrow 2\boldsymbol{R_0}$
  else [if $(k_j = 1)$]
    $\boldsymbol{R_0} \leftarrow \boldsymbol{R_0} + \boldsymbol{R_1}$; $\boldsymbol{R_1} \leftarrow 2\boldsymbol{R_1}$
return $\boldsymbol{R_0}$
```

Figure 3.8: Montgomery Ladder for Montgomery curves

equal to $\oplus$ defined the Section before). Because the computations can be carried out with the $x$-coordinates only, a lot of multiplications in $GF(q)$ are saved, resulting in an algorithm faster than other classical algorithms. Additionally, fewer memory is required since the $y$-coordinates need not to be handled during the computation of $\mathbf{x}(kG)$. When dealing with digital signatures, however, we need to know the sign of the $y$-coordinate of $kG$, $\mathbf{y}(kG)$. In this case, it can be efficiently computed at the end of the algorithm, from the $x$-coordinates of $kG$ and $(k+1)G$, which is contained in the register $R_1$.

The typical problem in ECC is the computation of a scalar multiple $kP$, where $k$ is the secret key. Anyone who observes the time taken by the ladder can deduce the position of the top bit set in $k$, since this position dictates the number of steps of the ladder. One fix is to always arrange for $k$ to have a fixed top bit, for example allowing only values of $k$ where the highest bit is always set to 1. Montgomery Ladder can also be parallelized (see, for example, [13]).

The most known implementation attack to Montgomery Ladder is the *side-channel attack*. A side-channel attack is any attack based on information gained from the implementation of a computer system, rather than weaknesses in the implemented algorithm itself. Timing information, power consumption, or even sound can provide an extra source of information, which can be exploited. The Montgomery Ladder is highly regular. Whatever the processed bit is, there is always a multiplication followed by a squaring (an addition and a doubling in the case of elliptic curves). The Montgomery Ladder can be implemented to prevent a given side-channel attack. Of course, this protection against simple sidechannel attacks do not ward off other kind of attacks. For other kind of attacks against Montgomery Ladder see, for example, [13].

## 3.5 Curve25519

In this Section, we do a detailed explanation of the curve chosen from WhatsApp, Curve25519. To ease the notation, we call $F_q$ the field $GF(q)$. We have already seen that the curve is

$$E : y^2 = x^3 + 486662x^2 + x,$$

defined over $F_q$ with $q = 2^{255} - 19$; $q$ is a prime. The curve was introduced from Bernstein. First of all, observe that $B = 1 \neq 0$ and that $A = 486662$ is a number chosen in a way that $A^2 - 4 \neq 0 \mod q$, so the curve is nonsingular. Moreover, $A^2 - 4$ is not a square. It is a Montgomery curve and we have already proved that it is birationally equivalent to the twisted Edwards curve

$$486664x^2 + y^2 = 1 + 486660x^2y^2.$$

Let's now investigate how it is used in cryptography.
The $x$- coordinate of the base point $B$ is $x = 9$. This point generates a cyclic subgroup whose order is the prime $p_{25519} = 2^{252}$+277423177773723535358519 37790883648493. The primality of $p_{25519}$ is essential because there is an attack which works with points with non-prime order that an attacker can use to save time in the computation of discrete logarithms. The order of the

point is really large, so it's possible to generate a lot of different and valid key pairs (i.e., a private key and the associated public key). The cofactor $c$ is equal to 8, so the order of the curve is $8p_{25519}$, while it can be proved that the order of the twist is $4r$ with $r$ prime.

The field where Curve25519 lives is $F_q = \mathbb{Z}_q$ . Since $F_q$ is a field, it is well known that there are exactly $\frac{q-1}{2}$ squares in $F_q$. Let's call $\delta$ the smallest number that is not a square in $F_q$. For the Fermat's Theorem, this happens if $\delta^{\frac{q-1}{2}} \equiv -1 \mod q \quad (\delta \neq 0)$.
Define $F_{q^2} := \mathbb{Z}_q \times \mathbb{Z}_q$. This structure has the following operations:

$$-(a, b) = (-a, -b)$$

$$(a, b) + (c, d) = (a + c, b + d)$$

$$(a, b)(c, d) = (ac + \delta bd, ad + bc).$$

It can be proved that $F_{q^2}$ is a commutative ring.
Define now $E(F_{q^2}) := \{\infty\} \cup \{(x, y) \in F_{q^2} : y^2 = x^3 + Ax^2 + x.\}$, with the convention $-\infty = \infty$. This curve has the classic formulas for addition, doubling and inverse for the points of a Montgomery curve. $E(F_{q^2})$ is a commutative group under these formulas. Finally, define the function

$$\mathbf{x_0} : E(F_{q^2}) \to F_{q^2}$$

$$\mathbf{x_0}(x, y) = x, \quad \mathbf{x_0}(\mathcal{O}) = 0.$$

Then, the following Theorem holds.

**Theorem 3.5.1.** *Let $q$ be a prime with $q \geq 5$. Let $A$ be an integer such that $A^2 - 4$ is not a square modulo $q$. Define $E$ as the elliptic curve $y^2 = x^3 + Ax^2 + x$ over the field $F_q$. Define the function $\mathbf{x_0}$ like above. Let $n$ be an integer and let $r \in F_q$. Then, there exists a unique $s \in F_q$ such that $\mathbf{x_0}(nQ) = s$ for all $Q \in E(F_{q^2})$ such that $\mathbf{x_0}(Q) = r$.*

Since we are talking about cryptography, we can see inputs and outputs of Curve25519 as sequences of bytes. By definition, the set of bytes is $\{0, 1, \ldots, 255\}$. There is a bijection between the two sets

$$\{0, 1, \ldots 2^{256} - 1\} \to \{0, 1, \ldots, 255\}^{32}$$

$$s \to \bar{s} = \left(s \mod 256, \left\lfloor \frac{s}{256} \right\rfloor \mod 256, \ldots, \left\lfloor \frac{s}{256^{31}} \right\rfloor \mod 256\right),$$

where the second set contains all 32-bytes strings.
A Curve25519 public key is a 32 bytes string, so it belongs to the set $\{0, 1, \ldots, 255\}^{32} = \{\bar{r} : r \in \{0, 1, \ldots, 2^{256} - 1\}\}$. We have already observed the $x$-coordinate of the base point $B$: it is a public point, so it belongs to this set, as it can easily be verified.
A Curve25519 secret key instead belongs to the set $\{\bar{n} : n \in 2^{254} + 8\{0, 1, \ldots, 2^{251} - 1\}\}$. This means private keys are integers $n$ with $2^{254} \leq n \leq 2^{255}$ and $n \equiv 0 \mod 8$. This particular choice is done to avoid two attacks:

- it sets the most significant bit to 1, getting rid of one of the Montgomery Ladder problem. In this way, the loop always has the same amount of iterations, and no timing information can accidentally be leaked by the variable iteration count;

- all keys are chosen $\equiv 0 \mod 8$ so no information about the secret key is leaked in the case of an active small-subgroup attack: in fact, we know that $c = 8$, so the order of the curve is a multiple of 8. This means there are some remaining points with smaller order. Suppose the following scenario: $A$ and $B$ wants to get a shared key. Their secret keys are $a$ and $b$. Let $G$ be the base point of Curve25519 $= (9, \ldots)$. We already know how they can derive a shared key $abG$. An active attacker could replace Bob's message $bG$ with a point of order 8 and be able to find $a \mod 8$ by inspecting following messages. When every valid secret key is $\equiv 0 \mod 8$, the attacker can't do nothing with this attack.

We can observe that both keys have always length 32 bytes = 256 bits. We can now define the map

$$\text{Curve25519} : \{\text{C25519 SecKeys}\} \times \{\text{C25519 PubKeys}\} \to \{\text{C25519 PubKeys}\}$$

$$(\bar{n}, \bar{r}) \to \bar{s},$$

where $s \in \{0, 1, \ldots, 2^{255} - 20\}$ has the following property: it is the unique integer such that $s = \mathbf{x}_0(nQ)$ for all $Q \in E(F_{q^2})$ such that $\mathbf{x}_0(Q) = r \mod 2^{255} - 19$, as the prior Theorem implied. Using a single coordinate instead of the whole point makes public keys smaller without the expense of point decompression.

If an application uses Curve25519, it must generate independent uniform random secret keys for every user: large deviations from uniformity can eliminate all security. As the name suggests, the secret key must be kept secret: only the user must know it, and it can be used to compute the associated public key $Curve25519(\bar{n}, \bar{9})$ or a shared secret Hash $H(Curve25519(\bar{n}, \bar{r}))$ for a given $\bar{r}$, where $H$ is a public hash function (e.g. SHA-256). The secret key must be reused with many public keys of other users and not thrown away after a single use. It is easy to design a safe function $H$ which works for a Curve25519 implementation: it needs less security of a safe encryption cipher to stop all known attacks. An attacker, of course, could have access to all informations about public keys of the users and to encrypted messages, which are protected thanks to a secret-key cryptosystem $C$, where the keys for $C$ are the shared-secret hashes $H(Curve25519((\bar{n_i}, \bar{r_j})))$ for various users $i$ and $j$.
The attacker could also generate many public keys $r'$ and, using $r'$ in the Diffie-Hellman protocol, see messages protected by $C$ where the keys for $C$ are of the form $H(Curve25519(\bar{n_i}; r'))$. In this case, the security depends on $C$ and not on Curve25519.

Computations on Curve25519 rely on fast $x$-coordinate scalar moltiplication and so on the Montgomery Ladder. The curve was chosen as a Montgomery curve to be able to do extremely fast $x$-coordinates operations. Curves of this shape have order divisible by 4, requiring a marginally larger prime for the same conjectured security level, but this is outweighed by the extra speed of curve operations. $\frac{A-2}{4}$ is chosen as a small integer, to speed up the multiplication by this fraction; this has no effect on the conjectured security level. It has been shown that this curve is more than twice faster than other curves and have the same conjectured security level.

As of August 2019, Curve25519 (and Curve448 too) are not standardized, but NIST will add them soon into their list of optimal curves to use in cryptography.

## 3.6 EdDSA signature schemes

In this last Section, to avoid confusion, we denote with coordinates $(u, v)$ the points of a Montgomery curve (e.g. Curve25519, the curve we use for

the description, even though the scheme remains valid for any Montgomery curve) and with coordinates $(x, y)$ the points of the associated (twisted) Edwards curve.
Edwards-curve Digital Signature Algorithm (EdDSA) is defined, like the name suggests, on twisted Edwards curves, where a public key is a compressed point consisting of a twisted Edwards $y$-coordinate and a sign bit $s$ which is either 0 or 1. The sign is 0 if the $x-$coordinate is the "positive" value, 1 otherwise (of course, if we are working in a finite field $GF(p)$ we only have positive values between 0 and $p-1$, but if we take, for example, $x = 3$ then we call $x = 3$ the number with $s = 0$, $x = -3 = p - 4$ the number with $s = 1$). Any efficiently computable birational equivalence preserves ECDLP difficulty, so the difficulty of computing ECDLP for Curve25519 immediately implies the difficulty of computing ECDLP for the associated twisted Edwards curve. The associated curve is also called Ed25519.
To ease the comprehension of the paragraph, we summarise here the 11 parameters which will be used during the Section:

- $B$ is the base point of Ed25519;
- $I$ is the identity point of Ed25519 $= (0, 1)$;
- $p$ is the prime of the field where we work, i.e. $p = 2^{255} - 19$;
- $q = 2^{252} + 27742317777372353535851937790883648493$ is the order of the base point $B$;
- $c = 8$ is the cofactor;
- $d = -\frac{121665}{121666}$ mod $p$ is the twisted Edwards curve parameter;
- $A = 486662$ is the Montgomery curve constant;
- $n = 2$ is a non-square integer modulo $p$ (the smallest number with this property for Curve25519);
- $|p| = \lceil log_2(p) \rceil = 255$;
- $|q| = \lceil log_2(q) \rceil = 253$;
- $b = 8\lceil \frac{|p|+1}{8} \rceil = 256$ is the bitlength for the encoded point.

We'll also use some functions written in pseudocode to explain the various steps of the algorithm.

Since a birational map between the two curves always exists, we can convert the $u$-coordinate of a point on the Montgomery curve to the $y$-coordinate of the equivalent point on the twisted Edwards curve. We call this function *u-to-y*. For Curve25519, the map is

```
Function $u\text{-}to\text{-}y(u)$
$y = \frac{u-1}{u+1} \mod p$;
return $y$
```

The function *convert* takes as input the $u$-coordinate and gives as output the point $P$ of the twisted Edwards curve. Before the conversion, the function masks off the excess high bits of $u$. This is done to preserve compatibility with point formats that reserve a sign bit for the use in other protocols. $s$ is always set to 0: since the function only takes the $u$-coordinate, it can't distinguish between the two possibilities for the twisted Edwards sign bit. The pseudo code of the *convert* function is the following (we denote $P_y$ the $y-$coordinate of $P$ and with $P_s$ the sign of $P$):

```
Function $convert(u)$
$u_{masked} = u \mod 2^{|p|}$;
$P_y = u\text{-}to\text{-}y(u_{masked})$;
$P_s = 0$;
return $P$
```

With this notation, we have $B = convert(9)$, because 9 is the $u$-coordinate of the base point of Curve25519. To make private keys compatible with this conversion, we define a twisted Edwards private key as a scalar $g$ where the twisted Edwards public key $G = gB$ has a sign bit of zero. Instead, we allow a Montgomery private key $k$ to be a scalar with any sign.
The conversion is done as following: we compute the "product" $E = kB$ and we define the $y-$coordinate of $G$ as the $y$-coordinate of $E$, then we impose the sign to be equal to zero. Then, the corresponding private key $g$ is adjusted so that it is a number with $s = 0$. The algorithm is called

*calculate-key-pair* and the pseudocode is the following:

```
Function calculate-key-pair(k)
$E = kB$;
$G_y = E_y$;
$G_s = 0$;
if $E_s = 1${
    $g = -k \mod q$;
}
else {
    $g = k \mod q$;
}
return $G, g$
```

This function requires at every call the computation of the point $E$. To improve speed, $E$ can be saved elsewhere, so the computation is only done once and not repeated every single time an user needs a point conversion. $E$ can be public because it is protected by ECDLP.

From now, we represent an integer in bold ($\mathbf{x}$) if it is a byte sequence of $b$ bits that encodes the integer in little-endian form, i.e. it places the least significant byte first, while an elliptic curve point in bold ($\mathbf{P}$) encodes $P_y$ as an integer in little-endian form with length $b-1$, followed by a bit for $P_s$. We study two digital signatures based on EdDSA: XEdDSA and VXEdDSA. They both require the Hash function SHA-512. if $i \geq 0$ and $2^{|p|} - i - 1 > p$, we define the family of Hash functions

$$hash_i(\mathbf{x}) = hash(\mathbf{2^b} - \mathbf{i} - \mathbf{1} || \mathbf{x}).$$

The second condition on $i$ is necessary because this scheme doesn't work for any $p$ which is a Mersenne prime. Prepending a different constant for each function call, we get several independent secure functions derived from a single secure Hash function. This is known as domain separation. In this way, we can use the same Hash function over and over without changing the private key: we just need to change the value of $i$, and thanks to the avalanche effect every time we change a single bit we get a completely different digest.

Both signatures are randomized, because at every call of the function a new, random, sequence of 64-bytes $\mathbf{Z}$ is produced.

### 3.6.1 XEdDSA

XEdDSA is so called because it uses X25519, which is the elliptic curve Diffie-Hellman exchange done using Curve25519. If $A$ wants to send a message $M$ to $B$, she also adds a signature in this way:

- $A$ takes her Montgomery private key $k \mod q$ and computes $G$ and $g$ (public and private key on the Edwards curve, respectively) through the function *calculate-key-pair*;
- $A$ randomly generates a 64-bytes sequence $\mathbf{Z}$;
- $A$ computes the digest $l = hash_1(\mathbf{g}||\mathbf{M}||\mathbf{Z})$ and then computes $r = l \mod q$;
- $A$ computes the Edwards curve point $R = rB$;
- $A$ computes the digest $t = hash(\mathbf{R}||\mathbf{G}||\mathbf{M})$ and then she gets $h = t \mod q$;
- $A$ computes $s = r + hg \mod q$;
- The signature is the sequence $\mathbf{R}||\mathbf{s}$: it has $2b$ bits, $b$ for the encoding of the elliptic curve point $R$ ($b - 1$ bits for the integer and 1 bit for the sign) and $b$ bits for the integer $s$;

The other user $B$ can verify the signature with the following verification procedure:

- $B$ takes $A$'s Montgomery public key $\mathbf{u}$ (byte sequence of $b$ bits), retrieves the plaintext $\mathbf{M}$ from the ciphertext he got from $A$, together with the signature $\mathbf{R}||\mathbf{s}$;
- He gets the corresponding number $u$, the $y$-coordinate of $R$ and the corresponding number $s$;
- $B$ does some preliminary computations: if $u \geq p$ or if $R_y \geq 2^{|p|}$ or if $s \geq 2^{|q|}$, he stops the procedure;

- $B$ computes the point $G$ of the Edwards curve corresponding to the coordinate $u$ with the function *convert* and check if the point is really a point of the curve: if it isn't, he aborts the procedure;
- $B$ finds the digest $t = hash(\mathbf{R}||\mathbf{G}||\mathbf{M})$ and then he computes $h = t \mod q$;
- $B$ computes the Edwards curve point $R_{check} = sB - hG$;
- $B$ compares the bytes of $\mathbf{R}$ and $\mathbf{R_{check}}$: if they are the same, he accepts the signature, otherwise he aborts the procedure.

This works because

$$R_{check} = sB - hG \Rightarrow R_{check} = sB - hgB = (s - hg)B = rB = R.$$

Let's see why the randomization is important. Suppose a deterministic procedure is followed, where the same nonce $r$ is used multiple times for different messages. Consider two XEdDSA signatures $\mathbf{R}||\mathbf{s_1}$ and $\mathbf{R}||\mathbf{s_2}$. By the definition of $s$, we have the following system of two equations:

$$\begin{cases} s_1 = r + h_1 g \mod q \\ s_2 = r + h_2 g \mod q \end{cases} \quad (h_1 \text{ and } h_2 \text{ change because they depend on } M).$$

This sistem has only two unknowns, $r$ and $g$, because any person ($B$ can as well) can compute $h$ and already has $s$ if it is in possess of the signature. This completely destroys XEdDSA, because we can efficiently compute the private key $g$ solving the system; an inverse is required but we know the extended Euclidean algorithm is efficient. If $r$ is instead obtained from the same Hash function, but without $Z$, the probability that different plaintexts give the same $r$ is small. However, if the same message is signed repeatedly, there is a sort of bug which affects the calculation of $h$, which could cause this to happen.

### 3.6.2 VXEdDSA

VXEdDSA extends XEdDSA transforming it into a Verifiable Random Function (VRF). VRF is a pseudo-random function that provides publicly verifiable proofs of its output's correctness. The VRF output for a given message and public key is indistinguishable from random to anyone who has not seen a VXEdDSA signature for that message and key. Using the VRF output and the public key, everyone can check that the value was indeed computed correctly, yet this information cannot be used to find the secret key.

VXEdDSA requires mapping an input message to an elliptic curve point. ECC protocols naturally send elliptic-curve points in the clear as long-term public keys, ephemeral public keys, etc. These points, even in compressed form, are obvious: they are easy to distinguish from uniform random strings. For example, an attacker can check if these sequences are squares modulo a prime. This has chance 1/2 of occurring for a uniform random string, but if it occurs repeatedly then the attacker is reasonably confident that the user is sending public keys. This is solved using the *Elligator2 map* and Hash functions. Elligator2 maps an integer $r$ in some $u$ for which $u^3 + Au^2 + u$ has a square root $v$ modulo $p$. We need the following Lemma.

**Lemma 3.6.1.** *If $u_1$ and $u_2$ are integers modulo $p$ such that $u_2 = -A - u_1$ and $\frac{u_2}{u_1} = nr^2$ for any $r$ and for a fixed nonsquare $n$, then the Montgomery curve $v^2 = u^3 + Au^2 + u$ has a solution for $u = u_1$ or for $u = u_2$.*

*Proof.* Let's define $w_1 = u_1^3 + Au_1^2 + u_1$ and $w_2 = u_2^3 + Au_2^2 + u_2$. We want to prove that either $v^2 = w_1$ or $v^2 = w_2$ has a solution (modulo $p$). Let's compute $w_2/w_1$ :

$$\frac{w_2}{w_1} = \frac{u_2^3 + Au_2^2 + u_2}{u_1^3 + Au_1^2 + u_1} = \frac{u_2(u_2^2 + Au_2 + 1)}{u_1(u_1^2 + Au_1 + 1)}.$$

By hyphotesis, $u_2 = -A-u_1$: substituting this value in the prior expression, we get

$$\frac{w_2}{w_1} = \frac{(-A - u_1)(A^2 + u_1^2 + 2Au_1 - A^2 - Au_1 + 1)}{u_1(u_1^2 + Au_1 + 1)} =$$

$$= \frac{(-A - u_1)(u_1^2 + Au_1 + 1)}{u_1(u_1^2 + Au_1 + 1)} == \frac{-A - u_1}{u_1} = \frac{u_2}{u_1} = nr^2,$$

where the last passage follows by the other hyphotesis. Now, we distinguish between two cases:

- Case $r = 0$: it must be $w_2 = 0$, and 0 is always a square modulo $p$;
- Case $r \neq 0$: in this case, since $n$ is not a square by hyphotesis, then $nr^2$ is not a square and so $\frac{w_2}{w_1}$ is not a square. By the Euler's Theorem,

we have $\left(\frac{w_2}{w_1}\right)^{\frac{p-1}{2}} \equiv -1 \mod p$ . This implies that one number between $w_1$ and $w_2$ is a square. In fact, if both of them are squares, then by the Euler's Theorem we have ${w_1}^{\frac{p-1}{2}} \equiv 1 \mod p$ and ${w_2}^{\frac{p-1}{2}} \equiv 1 \mod p$, so the ratio equals 1, which is against the hypothesis. The same reasoning can be done if both quantities are equal to $-1 \mod p$. The only possibility left is that one quantity equals $1 \mod p$, while the other quantity equals $-1 \mod p$: we can easily conclude that $w_1$ or $w_2$ is always a square modulo $p$.

□

We can find an exact expression for $u_1$ and $u_2$. In fact, by the hyphotesis of the Lemma, we have

$$\begin{cases} u_2 = -A - u_1 \\ \frac{u_2}{u_1} = nr^2 \end{cases}$$

Solving the system, we get

$$u_1 = -\frac{A}{1+nr^2}, \quad u_2 = -\frac{Anr^2}{1+nr^2}.$$

So, given $r$, we can compute $u_1$ and $u_2$ and then we can use the Legendre symbol to find which number is a square.

*Remark* 3.6.2. Let $p$ be an odd prime number. An integer $a$ is a quadratic residue modulo $p$ if it is congruent to a perfect square modulo $p$; otherwise, we say that $a$ is a quadratic nonresidue modulo $p$. The *Legendre symbol* is a function of $a$ and $p$ defined as

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{if } a \text{ is a quadratic residue modulo } p \text{ and } a \not\equiv 0 \mod p \\ -1 & \text{if } a \text{ is a non quadratic residue modulo } p \\ 0 & \text{if } a \equiv 0 \mod p \end{cases}$$

This definition is equivalent to $\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \mod p$ with $\left(\frac{a}{p}\right) \in \{-1, 0, 1\}$. The following properties ease the computations of the Legendre symbol:

- $\left(\frac{a}{p}\right) \equiv \left(\frac{b}{p}\right)$ if $a \equiv b \mod p$;
- $\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right)$ (it is a completely multiplicative function of its top argument);
- $\left(\frac{x^2}{p}\right)$ is equal to 1 if $a \not| p$, and it is equal to 0 if $a | p$;

- if $q$ is an odd prime with $p \neq q$, we have the quadratic reciprocity law:

$$\left(\frac{q}{p}\right)\left(\frac{p}{q}\right) = (-1)^{\frac{p-1}{2}\frac{q-1}{2}}.$$

The Elligator2 function implements this map from any integer $r$ to an integer $u$ which is a square modulo $p$:

```
Function Elligator2(r)
$u_1 = -\frac{A}{1+nr^2} \mod p$;
$w_1 = u_1^3 + Au_1^2 + u_1 \mod p$;
if $w_1^{\frac{p-1}{2}} \equiv -1 \mod p$ {
    $u_2 = -A - u_1 \mod p$;
    return $u_2$
}
return $u_1$
```

We have to map a byte sequence $\mathbf{X}$ onto an Edwards point: we hash the byte sequence and parse the hash output to get a field element $r$ and a sign bit $s$. Through the Elligator2 map, we convert $r$ to a Montgomery $u$-coordinate. Then, the birational map converts the Montgomery $u$-coordinate to an Edwards point. Finally, we multiply the Edwards point by the cofactor $c$ to be sure the point belongs to the subgroup generated by the base point $B$. The *hash-to-point* function implements these steps.

```
Function hash-to-point($\mathbf{X}$)
$h = hash_2(\mathbf{X})$;
$r = h \mod 2^{|p|}$;
$s = \lfloor \frac{h \mod 2^b}{2^{b-1}} \rfloor$;
$u =$Elligator2$(r)$
$P_y =$u-to-y$(u)$;
$P_s = s$;
return $cP$
```

VXEdDSA signature is more convoluted. It takes the same inputs of XEdDSA but uses them in a different way. If $A$ wants to add a VXEdDSA signature, then:

- $A$ takes her Montgomery private key $k \mod q$ and computes $G$ and $g$ (public and private key on the Edwards curve, respectively) through the function *calculate-key-pair*;
- $A$ maps the byte sequence $\mathbf{G}||\mathbf{M}$ onto an Edwards point with the function *hash-to-point*. She gets the point $B_v$;
- $A$ computes the Edwards curve point $V = gB_v$;
- $A$ randomly generates a 64-bytes sequence $\mathbf{Z}$;
- $A$ computes the digest $l = hash_3(\mathbf{g}||\mathbf{V}||\mathbf{Z})$ and then computes $r = l \mod q$;
- $A$ computes the two Edwards points $R = rB$ and $R_v = rB_v$;
- $A$ computes the digest $t = hash_4(\mathbf{G}||\mathbf{V}||\mathbf{R}||\mathbf{R_v}||\mathbf{M})$ and then she gets $h = t \mod q$;
- $A$ computes $s = r + hg \mod q$;
- $A$ computes the digest $x = hash_5(\mathbf{cV})$ and then computes $v = x \mod 2^b$;
- The signature is the couple $(\mathbf{V}||\mathbf{h}||\mathbf{s}), \mathbf{v}$, where $(\mathbf{V}||\mathbf{h}||\mathbf{s})$ is a byte sequence of length $3b$ bits and $\mathbf{v}$ is a VRF output byte sequence with $b$ bits, formed by multiplying the $V$ output by the cofactor $c$.

The other user $B$ can verify the signature with the following verification procedure:

- $B$ takes $A$'s Montgomery public key $\mathbf{u}$ (byte sequence of $b$ bits), retrieves the plaintext $\mathbf{M}$ from the ciphertext he got from $A$, together with the first part of the signature $\mathbf{V}||\mathbf{h}||\mathbf{s}$;
- B gets the corresponding number $u$, the $y$-coordinate of $V$ and the corresponding numbers $h$ and $s$;
- $B$ does some preliminary computations: if $u \geq p$ or if $V_y \geq 2^{|p|}$ or if $h \geq 2^{|q|}$ or if $s \geq 2^{|q|}$, he stops the procedure;

- $B$ computes the point $G$ of the Edwards curve corresponding to the coordinate $u$ with the function *convert* and checks if the points $G$ and $V$ are really points of the curve; if at least one of them isn't, he aborts the procedure;
- $B$ computes $B_v = \textit{hash-to-point}(\mathbf{G}||\mathbf{M})$;
- $B$ checks if $cG = I$, if $cV = I$ and if $B_v = I$ and in that case he stops the algorithm;
- $B$ computes $R = sB - hG$ and $R_v = sB_v - hV$;
- $B$ computes the digest $t = hash_4(\mathbf{G}||\mathbf{V}||\mathbf{R}||\mathbf{R_v}||\mathbf{M})$ and then he gets $h_{check} = t \mod q$;
- $B$ checks if the bytes of $h$ and $h_{check}$ are the same: if they are not, he aborts the procedure;
- If $h = h_{check}$, $B$ computes the digest $x = hash_5(\mathbf{cV})$ and then computes $v = x \mod 2^b$;
- Finally, he checks if $\mathbf{v}$ equals the second term of the signature.

The verification algorithm works because

$$R = sB - hG = sB - hgB = (s - hg)B = rB$$

and

$$R_v = sB_v - hV = sB_v - hgB_v = (s - hg)B_v = rB_v.$$

The rest of the signature is always correct: he gets $V$ from the first part of the digital signature (and he also verifies it is the real $V$ with the VRF), he gets $M$ decrypting the associated ciphertext and $G$ is simply the public key of $A$ on the associated Edwards curve.

Both EdDSA schemes require two calls to the Hash function (the only exception is the EXdDSA verification). A priori, we have not limited the length of the message $\mathbf{M}$ to be processed, and this can be expensive if the message is really long. The easier solution would be the introduction of a cap to the maximal length of the message.
It's also important that signing algorithms are performed in constant time, to not give additional informations (e.g. this avoids the infamous timing

attack). This is easily achieved for most functions there, exluded for the ones with an "if" statement, which have to be treated carefully.

# Chapter 4

# The Signal Protocol

The Signal Protocol is a non-federated cryptographic protocol that can be used to provide End-to-End Encryption (E2EE) for voice calls, video calls and instant messaging conversations. The protocol uses, among other things, prekeys, Curve25519, AES-256 and HMAC-SHA256. At a high level, Signal is an asynchronous channel protocol, between an initiator Alice and a recipient Bob, with the help of a key distribution server which only stores and relays information between parties, but does not perform any computation. When Alice wants to send a message, she obtains Bob's keys from an intermediate server, and performs a protocol to compute a message encryption key. Signal's goals include end-to-end encryption as well as advanced security properties such as forward secrecy and future secrecy: if we haven't forward/future secrecy properties, then we are not sure that, if an identity key is compromised, past/future session keys remain safe from an attacker.

The Signal protocol can be roughly divided into four steps:

- **Registration:** Every user, when installing the application, generates the keys and independently registers her identity with a key distribution server and uploads various kinds of public keys;
- **Session setup:** Alice asks and gets from the server a set of Bob's public keys and uses them to setup a messaging session. The session lasts for a long time. This is known as the Extended triple DH key agreement protocol (X3DH), and it is described in Section 4.1;
- **Synchronous messaging:** this situation happens when Alice wants to send a message to Bob and has just received a message from him.

In this case, she exchanges a new Diffie–Hellman value with Bob, deriving a new shared secret and then she uses it to start a new chain of Message Keys. Each Diffie-Hellman operation is a step of the asymmetric ratchet algorithm;

- **Asynchronous messaging:** this situation happens when Alice wants to send a message to Bob but has not received a message from him since her last sent message. In this case, she derives a new symmetric encryption key from the symmetric ratchet (known also as Hash ratchet) algorithm.

Both synchronous and asynchronous messaging are handled thanks to the Double Ratchet algorithm, which is described in Section 4.2. Each message sent by an user is encrypted using a new Message Key, which attempts to provide a higher degree of forward secrecy and future secrecy.
Finally, Section 4.3 describes the Sesame algorithm, which is used by Signal to guarantee perfect messaging not just between Alice and Bob, but between multiple users and multiple devices.

It's worth noting that, while Signal code is open source, WhatsApp code is not. They claim to use a specific implementation of the Signal protocol, and advanced reverse engineering tests done from experienced cryptographers all conclude this is indeed true. All considerations which will be done in this Chapter and in the next one are based on these conclusions, on Signal documentation and on the official WhatsApp white paper.

## 4.1 Extended triple DH key agreement protocol

Extended triple DH key agreement protocol (X3DH) establishes a shared secret key between two parties who mutually authenticate each other based on public keys. It doesn't only allow the two parties to communicate, but it also allows a secure communication. X3DH protocol involves 3 parties: two users (Alice and Bob) and a trusted server. Their roles are the following:

- Alice: she wants to send to Bob some initial data using encryption and she wants to establish a shared secret key which may be used to communicate between them;

- Bob: he wants to allow parties like Alice to send encrypted data and to establish a shared key with him. Bob might be offline when he receives data from Alice. For this reason, Bob has a relationship with a server;

- The server: it can temporarily store messages from Alice to Bob which Bob can later retrieve, and it can also let Bob publish some data. The server will provide these data to other parties like Alice. The server must be trusted: a malicious server could cause communication between Alice and Bob to fail, for example it could manipulate their messages.

X3DH is designed for asynchronous settings where one user (let's say Bob) is offline but has published some information to a server. If Alice wants to contact Bob, she can use that information to send him encrypted data, and they can also establish a shared secret key for future communication. WhatsApp has decided the following parameters for the usage of X3DH:

- Curve25519 and X25519;

- The Hash function SHA-256;

- *info*, an ASCII string identifying WhatsApp.

The protocol additionally defines an encoding function *Encode(PK)*, where $PK$ is a public key for X25519. It encodes the public key into a byte sequence. The recommended encoding consists of some single-byte constant to represent the curve, followed by little-endian encoding of the $u$-coordinate. We'll also use the following cryptographic notation:

- $DH(PK1, PK2)$ represents a byte sequence which is the shared secret output from X25519 involving the private key associated to the public key $PK1$ and the public key $PK2$;

- $Sig(PK, M)$ represents a byte sequence which is an XEdDSA signature on the byte sequence $M$. It is created using together the byte sequence $M$ with the private key associated to the public key $PK$, and it can be verified thanks to $PK$;

- $KDF(KM)$ represents the output of the HKDF algorithm with inputs:
  - Input Key Material $F||KM$, where $F$ is a byte sequence containing 32 0xFF bytes (the number 255) and it is used for cryptographic domain separation together with the digital signature XEdDSA, while $KM$ is an input byte sequence containing the secret key $DH(PK1, PK2)$;
  - salt, a zero-filled byte sequence with length equal to the hash output length (= 32 bytes): we are using the default salt value here;
  - info, which is an ASCII string identifying the application, as described before.
  - the sequence is $L = 32$ bytes long.

Every user $U$ has the following public keys (of course, every public key has an associated private key):

- A *long-term* Curve25519 key $IK_U$, generated at install time. "Long-term" means they are static: they are not session keys, they are used multiple times and never refreshed. It is used to sign the Signed Pre Key and to compute the shared secret;

- A medium-term Curve25519 key $SPK_U$, generated at install time, signed by $IK_U$, and rotated on a periodic timed basis. When a new key is signed, it will replace the previous one. The user may keep the private key corresponding to the previous signed key around for some period of time, to handle messages which used it that have been delayed. After they are handled, $U$ should delete this private key for forward secrecy. This key is "medium term", so it is not deleted after a single protocol call: it is shared between multiple users who want to contact Bob. This means that even if there are no more one-time keys stored at the server (see the next point), the session will go ahead using only a medium term key. This key is also used to generate the master secret;

- A queue of Curve25519 keys $OPK_U^1, OPK_U^2, \ldots$ for one time use, generated at install time, and replenished as needed, e.g. when the server informs the user that the server's store of one-time prekeys is getting low. Like the previous keys, they are used to generate the master secret.

In a protocol run, X3DH uses five Curve25519 public keys:

- Two long-term Identity Keys, $IK_A$ and $IK_B$, for Alice and Bob respectively;
- Alice's *ephemeral* key $EK_A$. A cryptographic key is called ephemeral if it is generated for each execution of a key establishment process. In some (rare) cases, ephemeral keys are used more than once, within a single session (e.g., in broadcast applications);
- Bob's Signed Prekey $SPK_B$: "prekeys" are so named because they are essentially protocol messages which Bob publishes to the server prior to Alice beginning the protocol run. They are used to guarantee asynchronicity, they allow Alice to establish a session even if Bob is offline;
- One of Bob's Onetime Prekeys $OPK_B^i$. If no one-time keys are used, Alice would employ only Bob's medium and long term keys: this setting doesn't guarantee good forward secrecy. Adding this key allows the shared secret to be based on truly ephemeral key on both sides of the communication. One-time prekey private keys will be deleted as Bob receives messages using them.

After a successful protocol run Alice and Bob will share a 32-byte secret key $SK_{AB}$.

X3DH has essentially three phases:

- **Phase 1: Publishing Bob's keys to the server**
  At registration time, Bob (a WhatsApp user, with at least a device) publishes the following keys, called the "prekey bundle", to the server:
  - A "long term" Identity Key $IK_B$;
  - A "medium term" Signed Key $SPK_B$;
  - A set of "short term" OneTime prekeys $OPK_B^1, OPK_B^2, \ldots$;

  - His Prekey XEdDSA signature $Sig(IK_B, Encode(SPK_B))$. Bob will upload a new prekey signature when he changes $SPK_B$: the new key will replace the previous value.

  The WhatsApp server stores these public keys associated with the Bob's identifier. WhatsApp guarantees that all the keys are generated client-side, so its server should have no access in any moment to any of the Bob's private keys.

- **Phase 2: Sending the initial message**
  Alice wants to talk to Bob, which means they need to do a X3DH key agreement. Upon request the server provides her the Bob's "prekey bundle", which contains $IK_B$, $SPK_B$, $Sig(IK_B, Encode(SPK_B))$ and, if it exists, a one-time prekey $OPK_B$. If the server provides $OPK_B$ to Alice, it deletes that key from the server storage. Alice and Bob might first want to compare their identity keys through a QR code mechanism. Some additional information will be given in Chapter 5, but QR code's functioning is out the scope of the thesis. More informations can be found on [21].

  First of all, Alice verifies the XEdDSA signature and aborts the protocol if the verification algorithm fails. Instead, if the check has success, she generates an ephemeral key pair, with public key $EK_A$.
  If a onetime key is not provided, Alice computes

$$DH1 = DH(IK_A, SPK_B),$$

$$DH2 = DH(EK_A, IK_B),$$

$$DH3 = DH(EK_A, SPK_B),$$

$$SK_{AB} = KDF(DH1||DH2||DH3) = HMAC_{salt}(F||DH1||DH2||DH3).$$

  $SK_{AB}$ is the shared secret key, also called *master key.*
  Instead, if $OPK_B$ is provided, after the first three steps she additionally computes

$$DH4 = DH(EK_A, OPK_B),$$

  and then the shared secret key is

$$SK_{AB} = KDF(DH1||DH2||DH3||DH4) =$$

$$= HMAC_{salt}(F||DH1||DH2||DH3||DH4).$$

$DH1$ and $DH2$ provide mutual authentication: they are necessary to provide a secure encrypted channel between the users. $DH3$ and $DH4$ provide forward secrecy: the absence of $DH4$ is not optimal because it lowers forward secrecy, which is quite important in instant messaging applications. Moreover, the same message may be replayed to Bob and he won't refuse it: he could receive from Alice the same message two times. This can be resolved updating the one-time keys more rapidly, or with the double ratchet protocol, which will be described in the next Section. If the attack has success, Bob computes the same master key in different protocol calls. The ratchet protocol also provides a randomization of the encryption key, combining $SK_{AB}$ with a freshly generated DH output, to avoid this security fall. It also seems that the prekey signature of Bob is not necessary for mutual authentication and forward secrecy. This is not true, in fact if Alice doesn't check Bob's signature, a corrupted server could give to Alice malicious keys and then retrieve Bob's private keys from Alice's message.

The following diagram summarises these computations.

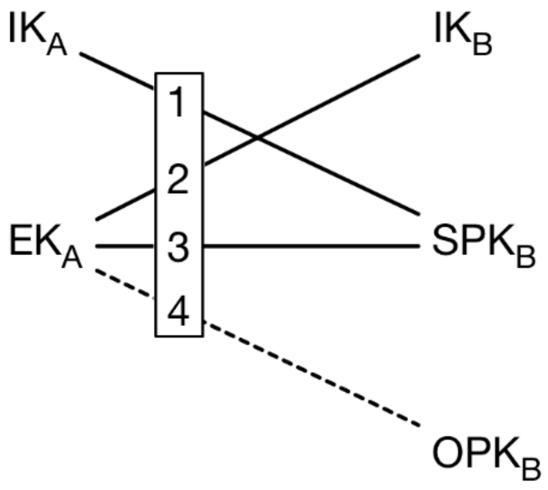


Figure 4.1: X3DH: computation of the shared key

After the computation of $SK_{AB}$, Alice deletes the ephemeral private key and every $DH$ output she got during the algorithm run.

Then, Alice calculates an Associated Data byte sequence $AD$ that contains identity information for both parties:

$$AD = Encode(IK_A)||Encode(IK_B).$$

Alice can add more personal informations to $AD$, like an ID or an username.
Finally, Alice sends to Bob an initial message containing $IK_A$, $EK_A$, the ID of the Bob's prekey used (if she used one) and an initial ciphertext encrypted with the AEAD encryption scheme using $AD$ as associated data and using as encryption key either $SK_{AB}$ or a key derived from $SK_{AB}$. Some examples of derived keys can be a KDF or a pseudorandom function PRF, which is a collection of efficiently-computable functions that emulate a random oracle in the following way: no efficient algorithm can distinguish (with significant advantage) between a function chosen randomly from the PRF family and a random oracle. For the AEAD scheme, WhatsApp uses AES256 in CBC mode and HMAC-SHA256.
Bob receives the message even if he is offline. Alice attaches these values to all messages she sends, until she receives an answer from Bob, because at that point she is sure Bob received $EK_A$.
The encrypted ciphertext has another role too: it serves as the first message for the double ratchet algorithm. After sending this message, Alice may continue using $SK_{AB}$ or keys derived from $SK_{AB}$ in the ratchet scheme to communicate with Bob.

- **Phase 3: Receiving the initial message** Bob retrieves $IK_A$ and $EK_A$ from the message sent from Alice and adds his identity private key, the private key corresponding to his signed prekey and the private key corresponding to his one-time prekey Alice used, if she used one. Bob repeats the same $DH$ and $KDF$ calculations Alice did to retrieve $SK_{AB}$, and then deletes the corresponding $DH$ values. After that, he computes $AD$ too, in the same way Alice did. Now Bob can decrypt the ciphertext using $SK_{AB}$ and $AD$. If the decryption fails, Bob aborts the protocol and deletes $SK_{AB}$. If the decryption has success, Bob deletes the private key corresponding to the one-time key used, for forward secrecy. If for some reason Bob's identity key is compromised, the deletion of private keys prevents the recovery of the old master keys. Bob may continue using $SK_{AB}$ or keys derived from $SK_{AB}$ within the post-X3DH protocol to further communicate with Alice.

Once the session is established, Alice and Bob do not need to rebuild a new session with each other until the existing session state is lost through an external event such as an app reinstall or a device change. X3DH is summarised in the Figure 4.2.

X3DH doesn't give to people who use the protocol a publishable proof of the contents of their messages or of their communication. An attacker could compromize the private key of Alice (or Bob) and start a conversation with Bob (Alice, respectively) pretending to be Alice. This limitation is intrinsic to the asynchronous setting.

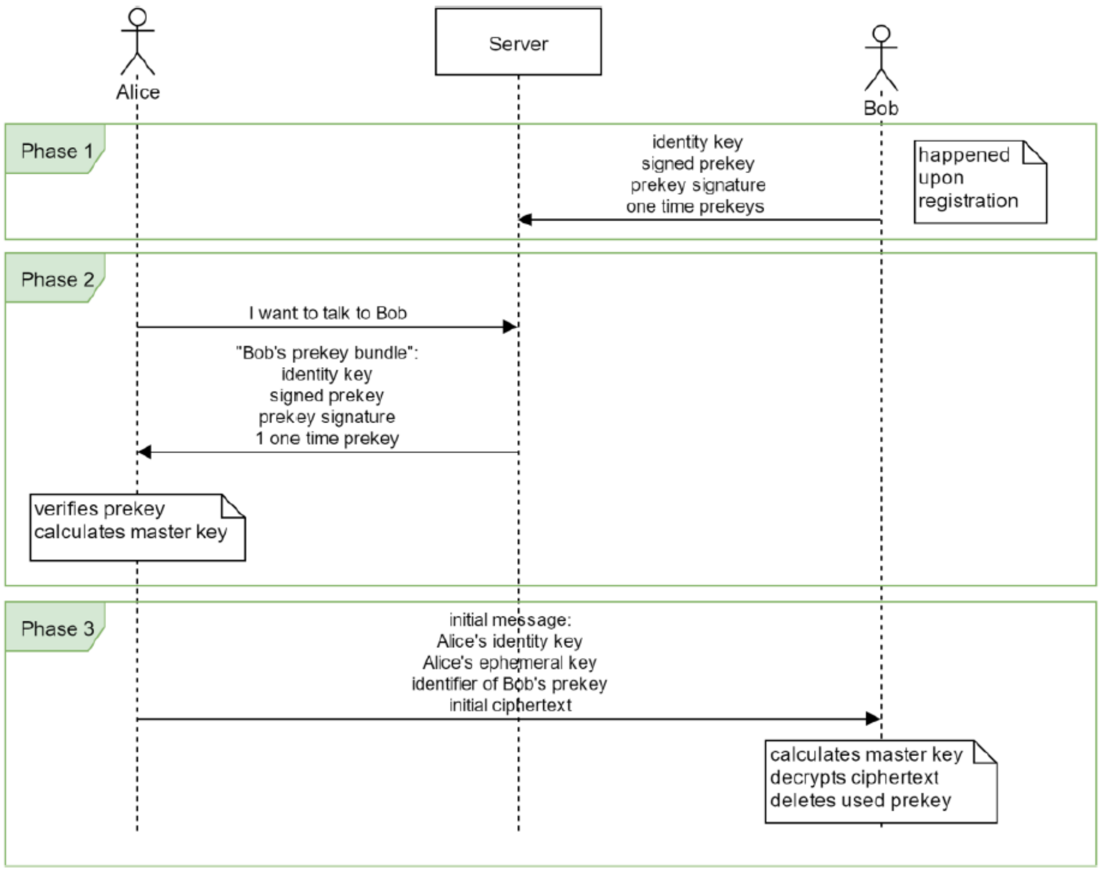


Figure 4.2: The X3DH protocol.

## 4.2 Double Ratchet

The Double Ratchet algorithm is used by two parties to exchange encrypted messages based on a shared secret key. After the agreement of a shared key through the X3DH protocol, Alice and Bob are now ready to exchange messages. Every time a new message is written, a new key is derived so that, if it is comprimised from an attacker, he can't retrieve old keys to read old conversations; moreover, the computations done to compute the new key is mixed into it so that noone can predict a new key starting from older ones. The algorithm is called *Double* Ratchet because it combines a Diffie-Hellman (so, asymmetric) key exchange ratchet with a symmetric-key ratchet.

The most important concept of the Double Ratchet algorithm is the *KDF chain*. We have already discussed KDF in Chapter 2: its output is a "strong" cryptographic key. In this context, strong means the KDF output is indistinguishable from random for any user who doesn't know the KDF key, i.e. KDF acts like a PRF; however, if the key is known or at least partially known, the KDF should still act like a secure cryptographic Hash. This is indeed always the case if the KDF is used with a secure Hash function like SHA-256.

**Definition 4.2.1.** Let's consider a standard KDF algorithm. If part of its output is used to build the strong cryptographic key and the other part of its output is used to replace the current KDF key for a new call of the algorithm, we have a KDF chain.

To be more clear about its functioning, the Figure 4.3 represents a KDF chain.
A KDF chain, like every pseudorandom generation algorithm, should satisfy the following properties (they were introduced in [27]):

- *Resilience*: every KDF output key is indistinguishable from random for every user/attacker without the KDF key, even if the KDF input is known;
- *Forward security*: if at some point an attacker gets a KDF key, he still has no way to distinguish between a past KDF output and random;

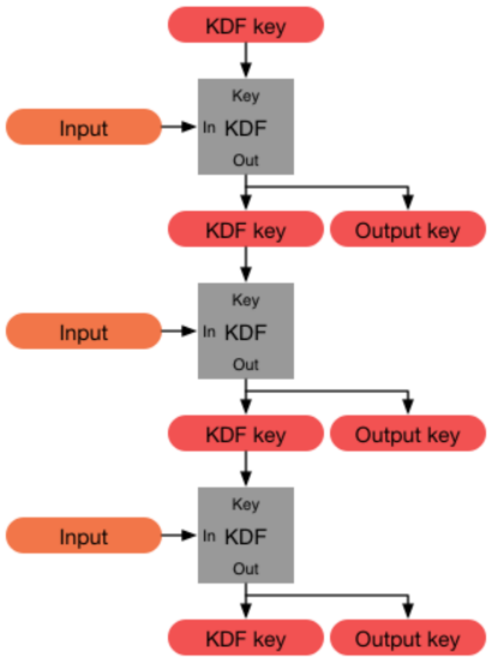


Figure 4.3: A KDF chain with 3 inputs and 3 outputs.

- *Break-in recovery*: if at some point an attacker gets a KDF key, he still has no way to distinguish between a future KDF output and random (provided that future inputs have sufficient entropy to do that);

Each user has a KDF key for three chains: a *root chain*, a *sending chain* and a *receiving chain*. Of course, if Alice sends a message to Bob, then Alice's sending chain is equal to Bob's receiving chain for that message. In the asymmetric ratchet, every time Alice and Bob exchange a message, they also generate a new Diffie-Hellman key pair and exchange the DH public key to derive a new DH secret: this output secret is the input of the root chain, and the output of the root chain is a new KDF key for both sending and receiving chains.
Every time a message is sent or received, the sending chain and the receiving chain advance ("ratchet" forward): their output key is used to encrypt/decrypt the message. This part of the algorithm is the symmetric ratchet.

Let's analyze the two ratchet's phases in detail.
In the symmetric ratchet, every message sent or received is encrypted and authenticated with an unique *Message Key*, which is the output of the

sending chain and of the receiving chain. A Message Key is 32 bytes long. The KDF keys for the sending and receiving chains are called *Chain Keys*: they are 32 bytes long and their purpose is the generation of Message Keys. The KDF inputs of these two chains are constants: they don't provide any entropy, so these two chains haven't the break-in recovery property. Indeed, these two chains are there just to be sure an unique Message Key is used for every message, and that this key can be freely deleted after the encryption or the decryption. So, a step of the KDF algorithm for the symmetric ratchet can be reassumed in this way:

- Inputs are the Chain Key $CK$ of the earlier call of the algorithm as the KDF key, and a constant $C$ as the KDF input;
- Outputs are a new Chain Key $CK$, which will be used in the next round of the algorithm, and a Message Key $MK$. Message keys are not used again to compute new keys and for this reason they can be stored without affecting the security of new Message Keys. This is useful because it allows the management of lost or out-of-order messages, which will be analyzed later.

WhatsApp uses HMAC-SHA256 for the KDF algorithm. Since the input is a constant, this phase is usually referred as *Hash ratchet*. $CK$ is used to derive the onetime use $MK$ with the following formula:

$$MK = HMAC_{CK}(0x01) \tag{4.2.1}$$

Then, the Chain Key is updated with the next constant:

$$CK = HMAC_{CK}(0x02). \tag{4.2.2}$$

The second formula could seem an useless passage because from the first Hash we already get a new Chain Key together with the Message Key, but there could be correlation between the two keys and, since we are allowed to preserve the Message Key, an attacker could utilize this property to get the corresponding Chain Key as well. For this reason, we exploit Hash properties to "ratchet forward" the Chain Key. Two symmetric ratchet steps are represented in Figure 4.4.

The Hash ratchet has optimal forward secrecy: if for some reason a Message Key or a Chain Key is compromised, the attacker can't retrieve old

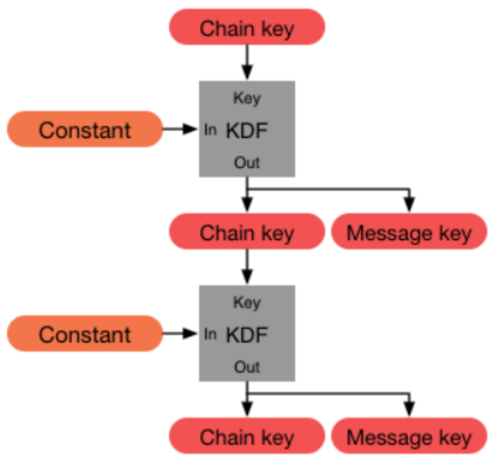


Figure 4.4: Two symmetric ratchet steps

messages since keys are ratched forward thanks to a Hash function, which is difficult to invert. However, the symmetric ratchet has not future secrecy: if an attacker steals the Chain Key of an user, then he can compute every future Chain Key and Message Key, and so he can decrypt every message, since the input of the KDF is a constant.

In the asymmetric ratchet, before sending a message every user generates a Curve25519 key pair: these keys are the current *ratchet key pair* of the user. Every exchanged message begins with a header containing an information about the current ratchet public key of the sender. When Alice receives Bob's new ratchet public key, she performs a new Diffie-Hellman ratchet step, which replaces her current ratchet key pair with a new Curve25519 key pair. When she answers the message, Bob will do the same. If an eavesdropper compromises the current ratchet private key, he can't do much with it because in a relatively short amount of time this key will be replaced with an uncompromised one and a new, unknown to the attacker, Diffie-Hellman output is generated.
Let's see how the Diffie-Hellman ratchet produces a shared sequence of DH outputs.

- Alice starts an algorithm's step with Bob's (ephemeral) ratchet public key and with the shared secret key $SK_{AB}$, obtained in the X3DH protocol, which is used as the initial Root Key $RK$. The Root Key is 32 bytes long and it is used to generate a new Chain Key. $SK_{AB}$ and

Bob's public key must be agreed before: the starting Bob's ratchet public key is his signed prekey $SPK_B$. Bob still doesn't know Alice's ratchet public key. We assume Alice is the first one sending a message: Bob doesn't send messages until he has received one from Alice;

- Alice does a DH computation with Bob's ratchet public key and with her ratchet (ephemeral) private key. The DH output is used as the KDF input of the root chain, whose outputs are a sending Chain Key $CK$ and a new Root Key $RK$;
- Alice sends a message $A1$ to Bob containing her ratchet public key. This message is usually the "initial ciphertext" described in the X3DH protocol;
- Bob receives the message and performs a Diffie Hellman ratchet step with Alice's ratchet public key and his ratchet private key (the key associated to $SPK_B$). Of course, this DH computation output equals the output got from Alice. He uses it as the KDF input of a root chain, whose outputs are a receiving Chain Key $CK$ and a new Root Key $RK$;
- Bob replaces his ratchet key pair and performs again a DH computation with Alice's public ratchet key and his new private ratchet key. He uses it as the KDF input of a root chain, whose outputs are a sending Chain Key $CK$ and a new Root Key $RK$;
- Bob sends a message $B1$ to Alice containing his new public ratchet key;
- Alice receives the message and performs a Diffie Hellman ratchet step with Bob's ratchet public key and her ratchet private key. Of course, this DH computation output equals the second output got from Bob. She uses it as the KDF input of a root chain, whose outputs are a receiving Chain Key $CK$ and a new Root Key $RK$;
- Alice replaces her ratchet key pair and performs again a DH computation with Bob's public ratchet key and her new private ratchet key. She uses it as the KDF input of a root chain, whose outputs are a sending Chain Key $CK$ and a new Root Key $RK$; Alice then sends a message $A2$ to Bob. From now, steps 4-8 are continuously repeated.

The Figure 4.5 reassumes the asymmetric ratchet from Bob's point of view.

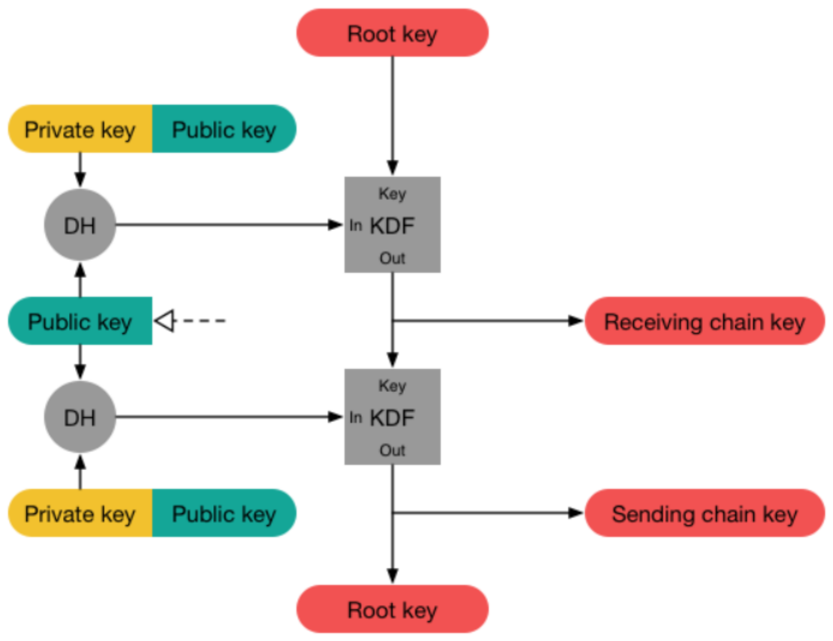


Figure 4.5: Bob's Diffie Hellman ratchet step

The asymmetric ratchet has optimal future secrecy properties, because every time a message is sent a new ephemeral key pair is generated, so if a message is compromised the attacker can't get the next one because he can't know the new randomly generated Curve25519 keys. However, forward secrecy leaves something to be desired: in fact, asynchronous chat sessions are extremely long-lived, as we have described during the X3DH protocol. If Alice sends a message to Bob, but he doesn't answer, the new Curve25519 key pair is not generated. If Alice sends to Bob multiple messages in a row, she always uses the same DH output: if this key is compromised, the attacker can derive all the messages sent with this key.

Perrin and Marlinspike, two Signal protocol founders, had the brilliant idea to put together the two ratchets, to combine future secrecy of the asymmetric ratchet and forward secrecy of the Hash ratchet, with as little of the negatives of both as possible.
Every time a new message is sent or received, the Hash ratchet is performed to derive a new Message Key. For example, when Alice sends her first message $A1$, she applies a symmetric-key ratchet step to the sending Chain Key

$CK$, obtaining a new Message Key $MK$. The new Chain Key is stored, but the Message Key and the old Chain Key can be deleted (the Message Key can be kept if needed). There can be many messages exchanged within the same DH ratchet round. Without the symmetric-key ratchet, as we have already observed, all messages encrypted based on the same DH key can be revealed. Every time a new Curve25519 public key is received, a DH ratchet step is performed prior to the symmetric-key ratchet to replace the Chain Keys. In this way, if the Chain Key is compromised it can't be used to decrypt future messages, since it is updated with a new, random Curve25519 key pair.

Let's analyze the steps of the Double Ratchet protocol from Alice's point of view. Suppose that, after the two messages $A1$ and $B1$, Bob sends her a new message $B2$, then she answers with three messages, $A2$, $A3$ and $A4$. In this case, Alice's sending chain will ratchet three steps, and her receiving chain will ratchet once. Then, if Bob writes two new messages $B3$ and $B4$ with his new public ratchet key and Alice answers with a single message $A5$, she performs two ratching steps in her receiving chain and then a single ratchet step in her sending chain.

The Figure 4.6 represents the tool situation we have just described: in the picture, Message Keys are labelled with the name of the message they encrypt or decrypt, and Bob's public keys are labelled with the message when they were first received.

If the message has been sent by Alice but still has not been received by Bob, she will see a single checkmark in the WhatsApp GUI; if Bob has received the message, Alice has two checkmarks instead. Alice can also send a message highlighting an old message: the content of the old message, together with its ID (every message has one; more informations are given in Section 4.3) are concatenated to the new message and all parts are encrypted. Messages can be deleted from the sender after they have been sent: the recipient is not able to read its content, receiving a WhatsApp notify ("This message has been deleted"), but this has not implications cryptographically.

Messages can arrive delayed, out of order, or can be lost entirely without

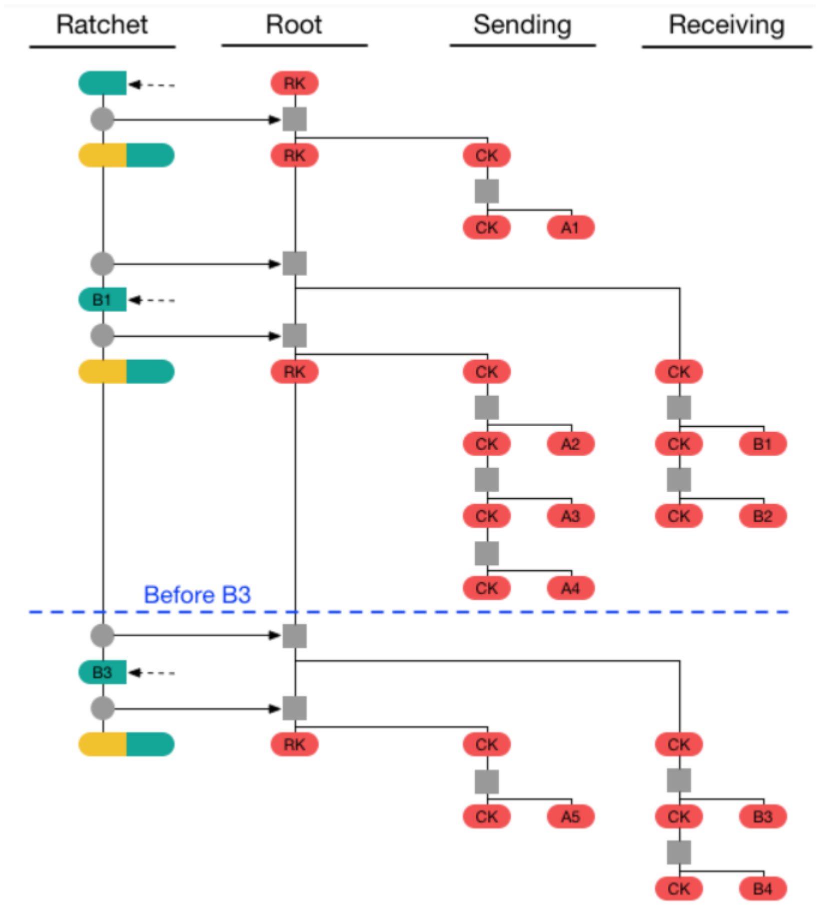


Figure 4.6: Alice's point of view in a message exchange

any implication as well. To keep track of them, in the header of every message, together with the new DH ratchet public key, there are two additional values: $N$, which indicates the number of the message in the current sending chain (the counter starts from $N = 0$) and $PN$, the length of the previous sending chain. $PN$ gives to the receiver the number of Message Keys of the last chain: by this approach, he can now store the Message Keys of these messages. We distinguish now between two situations, where we describe what happens if Alice is the initiator and Bob is the recipient:

- A new DH ratchet key pair is generated: then, in the header there is an info about the new public ratchet key. In this case, the number of skipped messages in the previous chain is

$$SkipSMS = PN - RecSMS,$$

where $RecSMS$ is the length of the current Bob's receiving chain. The received $N$ is the number of skipped messages in the new receiving chain, i.e. the one which starts after the DH ratchet. Bob must create it since he receives a new DH ratchet public key.

- If Alice keeps using the old public ratchet key, then

$$SkipSMS = N - RecSMS.$$

Consider the sequence of the previous example and suppose Alice doesn't receive $B2$ and $B3$ from Bob. She should have started a new DH ratchet with message $B3$, but obviously she can't now so she starts it with $B4$. The message $B4$ will have in its header $N = 1$ (it's the second message of that sending chain, since $B3$ is missing) and $PN = 2$ (the length of the previous chain, which contains $B1$ and $B2$). We are in the first case, since she receives a new public ratchet key. Then,

$$SkipSMS = PN - RecSMS = 2 - 1 = 1,$$

because the length of her current receiving chain is 1 (she only received $B1$) and $N = 1$, so she knows she also misses a message from the new receiving chain. Alice will then store Message Keys for $B2$ and $B3$ so they can be decrypted in the case they arrive.
The first message $A1$ is important because it contains the initial ciphertext for X3DH, and for this reason it is necessary this information doesn't get lost. We can't stop messages to go out-of-order, arrive late or never arrive, so, to handle this possibility, the recommentended pattern is for Alice to repeatedly send the initial message until she receives Bob's first Double Ratchet response message.

We discuss now how the algorithm is instantiated. Some pseudocode functions are used to ease the description of the protocol.
Every user tracks the following variables:

- $DHs$: a DH ratchet key pair. With the letter "s" we state this key pair is created by the user, and it is used to send messages;
- $DHr$ : a DH ratchet public key. With the letter "r" we indicate this public key is received when someone writes us a message, it is one of the variables in the header;
- $RK$: a 32 bytes Root Key;

- $CKs$: a 32 bytes Chain Key, used for the sending chain;
- $CKr$: a 32 bytes Chain Key, used for the receiving chain;
- $Ns, Nr$: Number of the message in the sending chain or in the receiving chain, respectively;
- $PN$: number of messages in the previous sending chain;
- $MKSKIPPED$: number of skipped Message Keys (= number of lost or out-of-order messages). They are indexed by $DHr$ and $Nr$.

Then, we define a constant $MAX\_SKIP$, which specifies the maximum number of Message Keys that can be skipped in a single chain. Setting a good number is not easy: it should be set high enough to allow lost or delayed messages, but it should also be low enough to avoid excessive receiver computations: in fact, a malicious sender could induce recipients to store large numbers of skipped Message Keys, causing a DoS attack. Finally, we need to define some functions:

- GENERATE_DH(): this function returns a new Diffie-Hellman Curve25519 key pair;
- DH($priv, DHr$): this function takes as input the private key $priv$ of the DH key pair $DHs$ and the DH public key $DHr$ and returns the output from the X25519 Diffie-Hellman calculation;
- KDF_RK($RK, DHout$): this function takes as input a 32 bytes Root Key $RK$, which is used as the HKDF salt, and the DH calculation output $DHout$ (used as the HKDF input key material), performs a HKDF step and returns a 32 bytes Root Key and a 32 bytes Chain Key. The recommended choice is HKDF with SHA-256, and it uses a WhatsApp specific byte sequence as $info$:

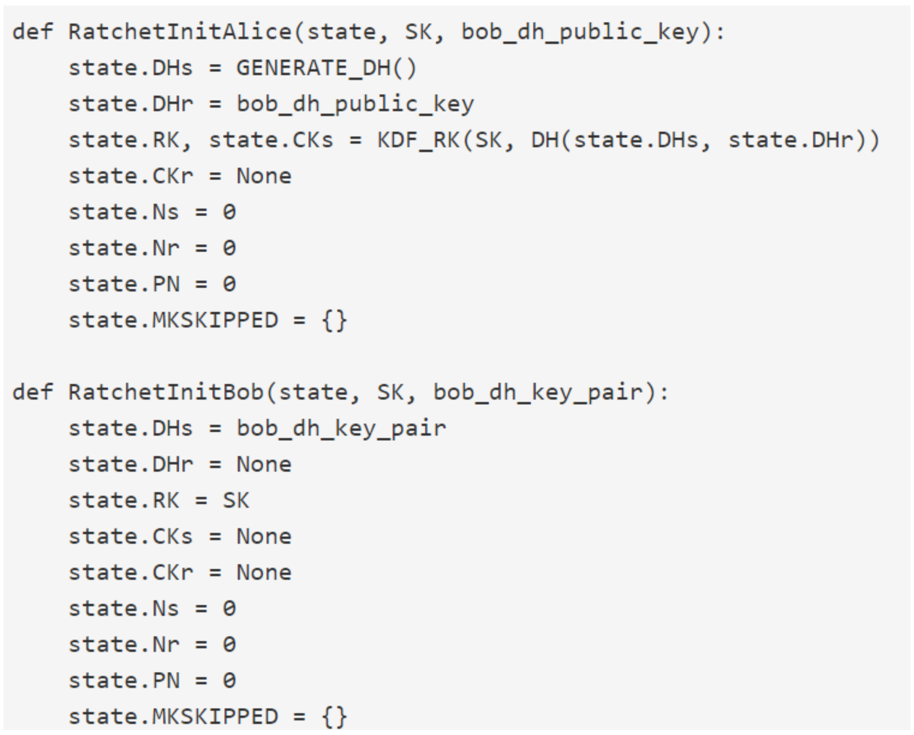
  $$CK, RK = HKDF_{RK}(DHout).$$

  Usually HMAC is the standard, but other algorithms can be used there as well;
- KDF_CK($CK$): this function takes as input 32 bytes Chain Key $CK$, together with some constant, and returns a 32 bytes Chain Key and a 32 bytes Message Key. HMAC with SHA-256 is recommended with $CK$ as HMAC key, while constants are the input key material;

- ENCRYPT($MK, M, AD$): it takes as input a Message Key, a plaintext $M$ and some associated data $AD$ and performs an AEAD encryption/authentication. It only encrypts $M$, while $AD$ is used for authentication and not included in the ciphertext. $AD$ is obtained during the X3DH protocol run. $MK$ changes at every iteration of the algorithm and for this reason it can be seen as a nonce. For this reason, the nonce in AEAD (see Chapter 2) can be fixed to a constant or derived from $MK$. WhatsApp uses AES in CBC mode with HMAC, because this scheme is resistant to attacks in the case a key is used two or more times in a row. For WhatsApp, the encryption scheme works as follows:
  - First, a HKDF with SHA-256 is used to produce a 80 bytes output, where the salt is the default value, the input key material is $MK$ and info is a WhatsApp specific byte sequence (it MUST be different from the info sequence used for the function $KDF\_RK$);
  - Then, the HKDF output is divided in this way: 32 bytes for an encryption key, 32 bytes for an authentication key and 16 bytes for an initialization vector IV;
  - At this point, $M$ is encrypted thanks to AES, with the encryption key and IV of the step before;
  - Finally, HMAC-SHA256 is performed with the authentication key and with the additional data as input. The output can be truncated up to 64 bits in some situations, to reduce message size.
- DECRYPT($MK, C, AD$): it takes as input a Message Key, a ciphertext $C$ and some associated data $AD$ and performs an AEAD deryption. The $AD$ used there is the same used in the ENCRYPT function. If the authentication fails, the function returns an exception. The nonce is treated like in the ENCRYPT function;
- HEADER($pk, PN, N$): this function takes as input the public key $pk$ of the DH key pair, the number of the message in the current sending chain $N$ and the length of the previous sending chain $PN$ and creates a new message header;

- CONCAT($AD, header$): this function takes as input the output of the function HEADER $header$ and some associated data $AD$. It encodes $header$ into a byte sequence and prepends, if possible, the $ad$ bytes sequence. If it's not possible, $AD$ is substituted by a length value. The output must always be a byte sequence $AD||header$.

The pseudocode of Figure 4.7 reassumes the start of the algorithm for both Alice and Bob. Here, $SK = SK_{AB}$ and in the class "state" we save all the informations of an user.

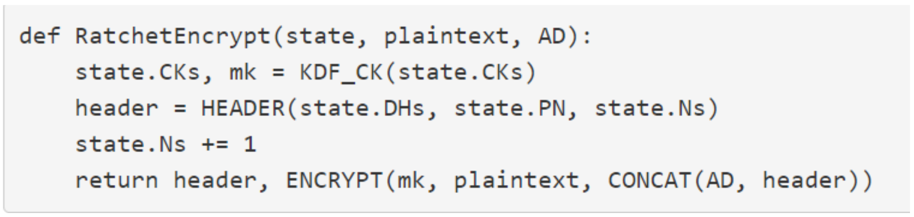


```
def RatchetInitAlice(state, SK, bob_dh_public_key):
    state.DHs = GENERATE_DH()
    state.DHr = bob_dh_public_key
    state.RK, state.CKs = KDF_RK(SK, DH(state.DHs, state.DHr))
    state.CKr = None
    state.Ns = 0
    state.Nr = 0
    state.PN = 0
    state.MKSKIPPED = {}

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

Figure 4.7: Double Ratchet initial conditions for both Alice and Bob

To encrypt a message, we call a function called $RatchetEncrypt$, which takes as inputs the class state, the plaintext and the additional data $AD$, computes a new Chain Key and a Message Key with the function $KDF\_CK$, computes the header with $HEADER$, updates the counter $Ns$ and returns the header and the output of the $ENCRYPT$ function. $AD$ is not used alone, but it is first concatenated with the header.
To decrypt a message, we call instead another function, $RatchetDecrypt$. It takes as input the class state, the header, the ciphertext and the associated data $AD$. This function does multiple actions: if the message received corre-



```
def RatchetEncrypt(state, plaintext, AD):
    state.CKs, mk = KDF_CK(state.CKs)
    header = HEADER(state.DHs, state.PN, state.Ns)
    state.Ns += 1
    return header, ENCRYPT(mk, plaintext, CONCAT(AD, header))
```

Figure 4.8: Double ratchet encryption

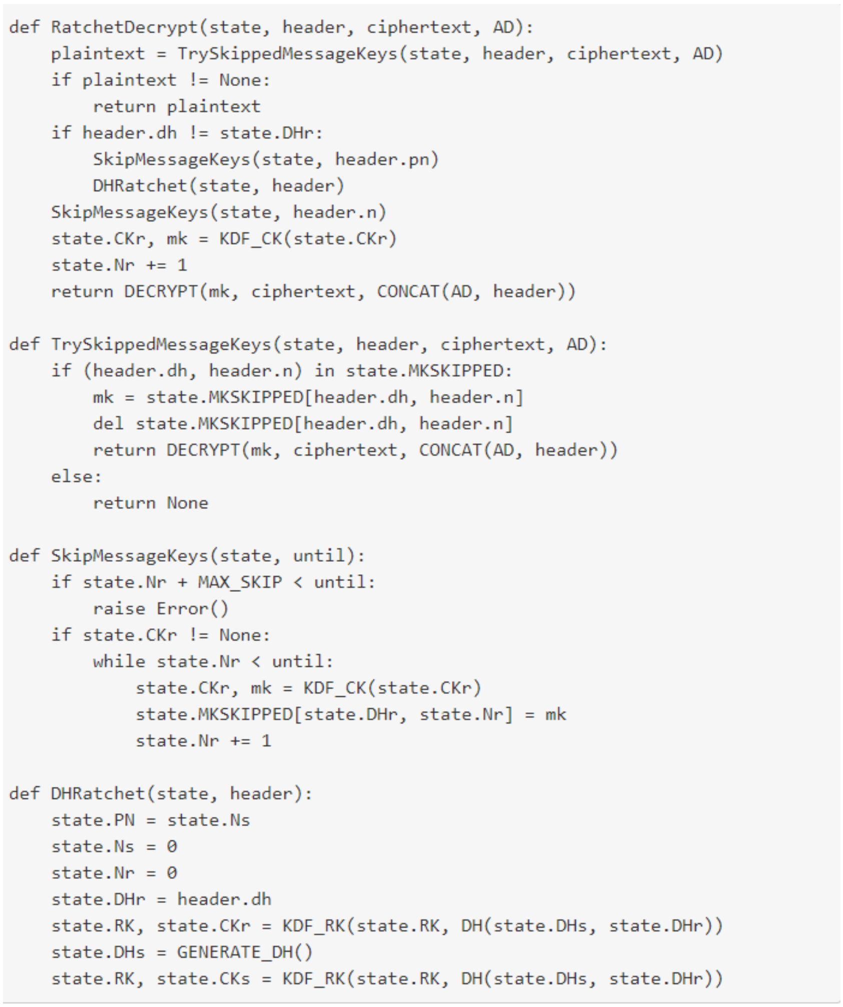
sponds to a skipped message (it is in $MKSKIPPED$), then this function decrypts the message, deletes the old Message Key, and returns the corresponding plaintext. If a new ratchet key has been received this function stores any skipped Message Keys and performs a DH ratchet step to replace the sending and receiving chains. Then, it stores any skipped Message Keys from the current receiving chain, performs a symmetric-key ratchet step to derive the Message Key and next Chain Key, and decrypts the message. The Figure 4.9 reassumes the decrypting function.

### 4.2.1 Double ratchet with header encryption

We have seen that every message has a header which contains three informations: the current public ratchet key of the sender, the number of messages $N$ in the current sending chain and the length of the previous sending chain $PN$. In the algorithm we just described, only the plaintext is encrypted, while the header is returned from the function $HEADER$ and nothing is done to hide its content. If an attacker intercepts a message, he can gain informations about the order of messages or the number of messages in the current chain.
A variant of the Double ratchet allows to handle this lack adding two keys. This adjustment is still not operative: neither Signal nor WhatsApp uses it. For this reason, only a brief description of the algorithm is given. The two keys are:

- A symmetric *Header Key* $HK$;
- A symmetric *Next Header Key* $NHK$.

The two keys are used for the sending chain and the receiving chain, but they differ, depending on which chain we are. So, in reality, there are four new keys. $HK$ is used to encrypt/decrypt a message header of the current



```
def RatchetDecrypt(state, header, ciphertext, AD):
    plaintext = TrySkippedMessageKeys(state, header, ciphertext, AD)
    if plaintext != None:
        return plaintext
    if header.dh != state.DHr:
        SkipMessageKeys(state, header.pn)
        DHRatchet(state, header)
    SkipMessageKeys(state, header.n)
    state.CKr, mk = KDF_CK(state.CKr)
    state.Nr += 1
    return DECRYPT(mk, ciphertext, CONCAT(AD, header))

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

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

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

Figure 4.9: Double ratchet decryption

chain, while $NHK$ is utilized for the same reason but in the new chain, which will be created if a new public ratchet key is received.

After a recipient receives a message, first of all she tries to decrypt the header with $HK$, $NHK$ or any header keys corresponding to skipped messages. One of these keys must work, so she gets informations about the chain. For example, if the key which works is $NHK$, then the recipient knows he must also perform a DH ratchet step to generate a new ratchet key pair. The DH ratchet step has now an additional phase: $NHK$ replaces $HK$, and a new sending $NHK$ is derived from the root KDF, while the sending chain $NHK$ is calculated from the root Chain Key and the DH values, while the receiving chain $NHK$ is negotiated directly as a shared secret. Alice and Bob must also initialize the protocol following these two rules, otherwise the two parties can't encrypt/decrypt:

- Alice sending $HK$ is equal to Bob's receiving $NHK$, so Alice's message $A1$ triggers a Bob Diffie Hellman step;
- Alice's receiving $NHK$ must be equal to Bob's (initial) sending $NHK$, so Bob's message triggers Alice's Diffie Hellman ratchet step.

Suppose now Alice is the initiator and Bob is the recipient. She has been initialized with Bob's ratchet public key, the shared secret $SK_{AB}$ which is used as the first Root Key, the sending $HK$ and the receiving $NHK$. Alice generates her ratchet key pair and updates the root chain to derive a new Root Key $RK$, the (sending) Chain Key $CK$, and the sending $NHK$. Alice then sends her first message $A1$: the header is encrypted with the initial shared sending $HK$. If Bob answers with a message $B1$, the header will be encrypted with the initial receiving $NHK$. Alice does a DH ratchet step, which will also shifts the header keys/ create new next header keys. Instead, if Bob sends a new message $B2$ with the same DH ratchet public key of $B1$, Alice only performs the symmetric-ratchet with the current sending header key.

The implementation is similar to the standard double ratchet: we just need to keep track of four additional variables (the header keys) and to define three new functions, two for the encryption/decryption of the header and one which modifies the $KDF\_RK$ function defined above, such that it generates a next header key too.

The Diffie Hellman ratchet algorithm is designed to give security against passive attackers who just observe encrypted messages after compromising a session, because it keeps generating new DH secrets. Despite this security, a compromise of secret keys can destroy the security of future communications. For example, the attacker can impersonate the compromised party, or could modify the user's Random Number Generator to predict the future ratchet keys. If an user suspects one of her keys or devices has been compromised, she must generate new keys.

### 4.2.2 Double ratchet and attachments

The double ratchet algorithm can also be used to end-to-end encrypt or decrypt message which contains attachments of any type, like video, audio, images or files. Let's see the procedure:

- Alice, the sender, generates an ephemeral encryption key of 32 bytes, an ephemeral authentication key of 32 bytes and a random initialization vector IV. This can be safely done exploiting the KDF properties, as described in the prior Section;
- Alice encrypts/authenticates the attachment like a normal message, using AEAD with AES in CBC mode, then she appends the HMAC of the ciphertext;
- Alice uploads the encrypted attachment to a Binary Large OBject (BLOB) store. BLOB is a collection of binary data stored as a single entity in a database management system, typically images and audio. Attachments are stored there and not on WhatsApp servers, otherwise we'd lose the E2EE property (WhatsApp never stores messages in its servers);
- Alice encrypts a normal message and sends it to Bob: it contains the encryption key, the HMAC key, a SHA-256 hash of the encrypted BLOB, and a pointer to the BLOB in the BLOB store;
- Bob decrypts the message, retrieves the encrypted BLOB from the BLOB store, verifies the SHA-256 hash of it, verifies the MAC, and decrypts the plaintext. Bob has received an attachment!

Attachments are sent to a BLOB store and not directly to the recipient because they could require some time to upload: in this way, the upload is done only once by the sender, then a simple message containing the data to retrieve the attachment is sent to every recipient. This also explains why the first time we send an attachment to one of our contacts, the upload requires some time, but if we send the same attachment to another contact, the upload is immediate.

## 4.3 The Sesame algorithm

The Sesame algorithm manages message encryption sessions in an asynchronous and multi-device setting. It is a generic algorithm which applies to any session-based message encryption algorithm, in particular it works with Double Ratchet and X3DH. We have seen that X3DH allows an user to create an encrypted session where she can communicate with another user even if the latter is offline; then, Double Ratchet updates their session keys after every message. On paper, these two algorithms are enough to perfectly manage the communication between Alice and Bob, while in practice there are still many problems which are not handled. For example:

- Alice and Bob can have multiple devices, so a single session is not enough. There must be a session between every Alice's device and every Bob's device, because both of them must receive the message or see the sent message in every device they can login. For this reason, there must also be sessions from a device to the other devices of the same user. Moreover, they can add or remove a device in every moment, so sessions must be added or removed in the same instant. Usually, a device is the phone of the user or a personal computer;
- Alice and Bob could start the X3DH protocol in the same moment, creating two different sessions. Double ratchet performs better if they use the same session to communicate, so they must agree on which session they have to use if this "accident" happens;
- Alice or Bob could decide to restore a backup, in which case the new session doesn't match anymore the session of the other user.

These three problems are the main cause of lost or out-of-order messages. Sesame manages all these requirements, defining the state of each device and the algorithms which utilize this state to exchange encrypted messages. The main idea is the following: each device remembers an active session for each other device it is communicating with, and uses this session to send to that device. Usually, every device has a single active session (and no inactive sessions) with every device it is communicating with: however, multiple sessions between the same devices can raise in the case one user restores a backup. In this case, when a message is received on an "inactive" (old) session, that becomes the new active session.
We need to do some assumptions on server, users and devices:

- **The server**: it always has in memory every user and every device. Moreover, it can store temporarily messages, until the other device receive them (see X3DH). We assume there is just one server, but of course any real app (WhatsApp included) has more than a server to keep track of everything: for example, there could be a server to handle users/devices and another server to manage mailboxes;

- **Users**: there is always at least an user which utilizes the application, and they can be added or deleted in any moment. Every user has a different UserID: usually, an username, or directly the phone number. Signal and WhatsApp use the phone number as the UserID. When an user is removed, his ID can be taken from a new user;

- **Devices**: each registered user has always at least a device, and he can add or remove a device in any moment. Each device has an unique DeviceID and an identity key pair (i.e., a public and a private key). If Sesame is used with X3DH, every device also has a signed prekey and a queue of one-time prekeys. A device can ask to the server informations about other users and devices and keep track of a state. This state could be deleted or rolled back if a backup occurs. Every device has a clock to track the passed time;

- **Mailboxes**: every device has a mailbox, which holds messages sent to that device: they are stored in the server. The message is deleted from the mailbox when it is received. If Alice (who has a device) sends a message to Bob (who has another device), the server stores in Bob's mailbox the message together with Alice ID and device. The recipient Bob can retrieve from his mailbox the message, the Alice's ID and the Alice's device used. Messages could be corrupted, attacked, delayed or out-of-order; moreover, if they don't arrive after a fixed time period, they are considered lost. We call this time period $MAXLATENCY$;

- **Sessions**: A session is indexed by an unique SessionID and it is used to encrypt and decrypt the messages; it is secret and it is stored in a device. If a message is encrypted in a session, it can be decrypted only in a matching session. Data in a session continuously change after a successful encryption/decryption (new keys are generated and old ones are deleted, for example). A device can create a new session in any time through the X3DH protocol, if the public key of the recipient device is given. All messages encrypted by the new session are initiation messages, and they contain in the unencrypted header informations about the sender's identity public key. When it is received from the recipient, the matching session is created. After the first decryption, the session becomes regular and it stops sending initiation messages.

Let's analyze how Sesame works. Each device communicates with multiple users, so it stores a list of UserRecords, indexed by their UserID. Each UserRecord has a list of DeviceRecords, indexed by their DeviceID. Each DeviceRecord may contain an active session or a list of inactive sessions. The list of inactive sessions must be kept ordered: in the head there is the last active session (now inactive), at the end of the list there is the oldest active session. It's important to note that a device stores UserIDs, but it does NOT store DeviceRecords for its DeviceIDs: the UserRecord enables a device to send a copy of each message to the other devices of the user. An UserRecord or DeviceRecord might be marked *stale*: this corresponds to a deleted user or device which is not deleted from the device to allow the decryption of delayed messages. The stale record contains a timestamp which saves the moment when it was marked stale: if the clock passes the fixed $MAXLATENCY$, the stale record can be deleted. The identity key pair is used for cryptographic authentication, and it is never replaced, unless the device is logically deleted and then readded with new values (we have already discussed about it, it's a long term key pair). Key pairs can be derived in two ways:

- *per-user identity keys*: all devices of the same user have the same identity key pair. In this case, the key pair is stored in UserRecord;
- *per-device identity keys*: each device of the same user has a different identity key pair. The key pair is stored in DeviceRecord. WhatsApp derives the key pairs with this technique.

Devices can modify their local state in various ways. For example, they can delete, if necessary, sessions, DeviceRecords or UserRecords. If a device remains without sessions or an user remains without devices, they are deleted as well. Sessions can also be added in a DeviceRecord: when this happens, it becomes the active session. The old active session goes into the inactive list. The list could have a number of maximum entries: if the number of sessions exceeds this number, they are deleted from the tail of the list. There

is only ONE active session for each device we are communicating with: if a session in the inactive list goes active again, the current active session goes in the list instead.
Finally, devices can receive as an input the three values UserID, DeviceID and PublicKey, and upgrade their records in this way:

- if the corresponding UserRecord doesn't exist or if it has a different public key value, a new record is added to handle it. The old one is deleted. The input public key is stored in this new record in the case of per-user identity keys;

- if the corresponding DeviceRecord doesn't exist or if it has a different public key value, a new, empty, record is added to handle it. The old one is deleted. The input public key is stored in this new record in the case of per-device identity keys. The device where this check is done does not add a new DeviceRecord for itself (i.e., if the inputs are UserID and DeviceID of the device which is doing the operation, nothing happens). A device can preps for encrypting to the tuple (UserID, DeviceID, public key): in this process, first of all the UserRecord and the DeviceRecord are marked stale, if necessary. Then, the device updates its record based on the tuple. Finally, if the relevant DeviceRecord doesn't have an active session, then the device creates a new session using the relevant public key for the DeviceRecord. The new session is inserted into the DeviceRecord.

Let's describe the Sesame sending process. The sending device wants to send a message $M$. Let's denote with $ID_1, \dots, ID_N$ the userIDs of the list of recipients linked to her device, plus her own UserID. To encrypt and to send $M$ to each linked user, for each $ID_i$ this algorithm is followed:

- the sending device checks if there exists a relevant non stale UserRecord for the user $ID_i$, then it searches and gets all active sessions (with respect to the sending device) for each non stale DeviceRecord in this UserRecord. The sending device encrypts $M$ (through the Double Ratchet) using the active sessions;

- $ID_i$ is sent to the server with the list of encrypted messages and the list of DeviceIDs of the user $ID_i$. The list of DeviceIDs indicates the recipient mailbox for each message. Lists are empty if no active session exists;

- The server checks if $ID_i$ exists and if it has the declared list of DeviceIDs. In this case, the server accepts the encrypted messages, which are sent to the mailboxes linked to the DeviceIDs;
- If something is wrong, the server rejects the messages and informs the sending device about what did not work. If the UserID doesn't exist, the sending device marks the UserRecord as stale and passes to the next user. If the DeviceID is old, the sending device gets informed about the new DeviceID and the new associated public key. Old devices are marked as stale, while new devices are prep for encrypting to the tuple (UserID, DeviceID, relevant public key).

If some error occurs in encrypting to an user, the sending device should avoid to communicate with the relevant UserRecord. It's up to the user to decide if it's better to stop the whole process or to continue the encryption with other users. There should also be a limit to the number of times the user tries to send a message to a recipient's mailbox, to avoid the possibility of bugged server or malicious attacks.

The other Sesame algorithm is the receiving process. The sending process has sent the encrypted message $C$ to the relative mailbox. Now, inputs are $C$, the sender's userID $uID_S$ and the sender's deviceID $dID_S$. The algorithm gets all the inputs from the server. Then, $C$ is decrypted in this way:

- If $C$ is an initiation message and the recipient device does not have a session that can decrypt the message, then additional steps are required:
  - the relevant public key is taken from the message header;
  - the device updates its record with $uID_S$, $dID_S$ and the relevant public key;
  - the device creates the new session with the initiation message and insert it into the DeviceRecord.
- If no session in the DeviceRecord can decrypt $C$, then $C$ is discarded;

- If the right session exists but it was inactive, it is activated, then $C$ is decrypted (through the double ratchet algorithm). If any error occurs in the procedure, including cryptographic errors, then it's better if the device discards all state changes and $C$, ending the decryption process.

If a device has been deleted or rolled back, some (valid) message could not be decrypted. To not lose any message, an user could save in her sending device a set of MessageRecords, indexed by an unique MessageID. The same message could be sent to more than a person. In this case, every message is treated like a different one, and will have its own MessageID. The MessageRecord stores the plaintext $M$, the recipient UserID and the SessionID where $M$ was encrypted.
If the recipient device has troubles in the decryption, it sends to the original sending device mailbox (i.e., the device which sent the "problematic" message) an unencrypted retry request with the MessageID of the message. The sending device also retrieves the UserID and the DeviceID of the recipient device. Then, the original sending device does a resending process:

- if the MessageID doesn't exist, the process stops;
- if the received UserID is not equal to the MessageRecord UserID, the process stops. There is NOT a control for the device, because if this situation occurs, the device could have been changed;
- the original sending device checks about the session. If the relevant DeviceRecord has not an active session, or if it has one but it is the same session located into the MessageRecord, a new session must be created. The original device asks the server about the relevant public key, then it preps for encrypting to the tuple (UserID, DeviceID, public key) if the DeviceRecord doesn't have an active session, otherwise, if the session exists but it matches the one in the MessageRecord, a new initiating session is created and inserted into the DeviceRecord, to avoid a resending process into an orphaned session.
- $M$ is encrypted in the new session of the relevant DeviceRecord;
- $C$ (the encrypted message) is sent to the server together with UserID and DeviceID of the recipient. The server stops the process if it hasn't saved that particular UserID or DeviceID in its memory;

- The server accepts the message, which is sent to the recipient's mailbox. The old MessageRecord is deleted, a new one for this new message is created.

A MessageRecord may be deleted after some time or if the plaintext has been erased from the sending device. WhatsApp uses delivery receipts to notify the successful decryption: they refer to some MessageID and notify the sender that the MessageRecord may be deleted. Devices should impose a limit on the number of times they're willing to resend a message, to avoid attacks like DoS.
For security, sessions might be replaced after some time.

### 4.3.1 Sesame's security

Let's briefly discuss the security of the protocol. Sesame relies on users authenticating each other before using it. They can use a QR code, as discussed in the X3DH Section. When dealing with a message, the sender could notice a change of the recipient's identity public key. This could mean that the UserID was deleted and taken from another user, a new device has been added, or someone is impersonating the recipient's device. In any of these cases, the users MUST perform again the authentication process and stop the sending, receiving or resending process.
The user with the compromised device must replace it and the associated key pair; then, it should notice every user he is in contact with of the change. In fact, suppose an attacker compromises Bob's device. Then, he can send a message to every UserID linked to Bob's device, impersonating him; or he could be a passive attacker, who just reads all messages. The attacker could also gain access to old plaintexts archived in the device. Finally, the attacker might try to use the keys for passive decryption, decrypting old communications or future communications. He could also try to reveal the device's state of some past moment (e.g. he can find an old backup) and then use this state to decrypt old messages. The security against passive decryption completely relies on the long term, medium term and ephemeral keys used during X3DH and double ratchet. A server could be corrupted too: it can send to the device what it wants to compromise new X3DH initial messages without a one-time prekey: if it owns the signed prekey's private key, it can also decrypt all messages sent during the cycle of the

medium-term key pair. This explains why signed prekey should be changed regularly.
There is a last kind of attack which can be performed against Bob: the server could impersonate other users with a contact with Bob (e.g., he could impersonate Alice). When Sesame uses X3DH, the server can use a compromised signed prekey with X3DH to create sessions with the target: in poor words, it can create a fake session, where Bob thinks he is communicating with Alice. Even then, the replacement of the signed key from time to time is necessary, or there will be risks until the session is deleted. The compromission of a private key is always possible, in the sense that no protocol can offer 100% security: anyway, every user should follow these recommendations because they highly mitigate the risk of an attack.

If the communication is authenticated but an attacker still manages somehow to impersonate Bob, he can't decrypt messages but he can get sender's UserID and DeviceID. He could also register to the server with Bob's parameters, effectively deleting Bob's account. The server must mitigate this attack, and it usually has two ways to do it: either DeviceID is assigned from the server, so they are random, or the server can require a code which requires the possession of Bob's private key too (with this attack, he does NOT have the private key, he is just impersonating Bob with his UserID and DeviceID).

We have seen how Sesame manages the deletion of old messages. If the clock is not perfectly synchronized, it could result in the recipient deleting the sessions too early and not being able to decrypt some delayed messages. In the other way, if clock errors prevent the recipient's clock from advancing, these older sessions might never be deleted. Clocks should be reliable and impossible to manipulate. They can be combined with other forms of security, to provide crossed checks.

Finally, some attention should be taken against messages which were not delivered to every user. This is a problem especially in group chats, where it's important every user receives the message. One option would be to add a function which committes messages to mailboxes only if sending succeeds for all users.

# Chapter 5

# Additional WhatsApp features

We have seen how messages can be exchanged between two users in an E2EE application. WhatsApp is much more than a simple instant messaging app between two people: it allows chat groups, voice and video calls, statuses and live location in an authenticated environment. Section 5.1 describes WhatsApp group chats, and it focuses on the algorithm as well as on the limitations of this protocol, like the lack of future secrecy. Section 5.2 describes the encrypted calls on the app, while Section 5.3 and 5.4 talk about features derived from group chats, like statuses and live locations. Section 5.5 tells something more on security and QR code, while Section 5.6 explores one of the extra applications of the latter, explaining how it can be used to establish a twin session of WhatsApp in the browser.
Finally, we give an idea in Section 5.7 and in Section 5.8 on how the communication between WhatsApp clients and WhatsApp servers works, on the importance of metadata and on how backups are treated.

## 5.1 WhatsApp group messages

WhatsApp offers to its users the possibility to have end-to-end encrypted messages not just between two people, but between multiple users as well. The Signal protocol doesn't work quite as well for group messaging, primarily because it's not optimized for broadcasting messages to many users. To handle groups, WhatsApp doesn't faithfully follow Signal and adds a key for every user, denoted Sender Key, which this member will use to encrypt all of her messages to the group.
Traditionally, unencrypted messenger applications used "server-side fan-out"

for group messages: with this method, if Alice wants to send a message to a group of users, she only has to transmit a single message, which is then distributed to every different group member by the server. This is different from the method “client-side fan-out,” where Alice would transmit a single message to every different group member herself.
Messages to WhatsApp groups are based on the pairwise encrypted sessions outlined in Chapter 4. The strategy chosen is the "server-side fan-out", but Signal provides also end-to-end encryption.

A WhatsApp group is identified by $ID_{gr}$, which contains the ID of the group creator (her phone number) and a timestamp. A group can contain up to 256 members, and the number of administrators (i.e., users which have the power to add or remove other users from the group, and so on) is a subset of the users in the group. The group creator is always an administrator. Although WhatsApp integrates X3DH, keys in groups are used very differently: instead of sending encrypted messages to each group member separately, each user generates a Chain Key for encrypting only her messages to the group. The key is then immediately sent to the other members of the group using the Double Ratchet algorithm. The group key is updated only with the symmetric ratchet : this is a significative difference with respect to the communication between just two users.
There is a distinction between the first message sent in a group from an user and every other message sent by the same user after that. Let’s see how it works in both cases: we suppose only text messages are exchanged, for simplicity. The first time a WhatsApp group member sends a message to a group, then:

- she generates a random 32 bytes Chain Key. This key is separated from the Chain Key used for direct messaging, the two chains have nothing in common. Every group has its own chain;
- she generates a random Curve25519 key pair: we denote this pair *Signature key.* The current signature key for the respective group is used to sign the ciphertext;
- she combines the 32 bytes Chain Key and the public key from the Signature Key into a *Sender Key* (for example they may just be concatenated);

- she individually encrypts the Sender Key to each member of the group, using the X3DH protocol.

So, each group participant has her own Sender key. Whenever a new member joins the group, it generates her own Chain key and a Signature key pair. The Sender key is distributed to all the group participants using the pairwise direct messaging. All the other group participants already have their own Sender key and they share it with the new participant through pairwise messaging as well. In this way, everyone keeps the Sender key of each other. Sender key is then decomposed to Chain key and public Signature key once it is exchanged.
Now, for all subsequent messages to the group:

- she derives a Message Key from the Chain Key through the sending chain, and updates the Chain Key, with formulas 4.2.1-4.2.2 and the Hash ratchet;
- she encrypts the message using AES256 in CBC mode and authenticates it using HMAC;
- she signs the ciphertext using its private Signature Key with XEdDSA;
- she transmits the single ciphertext to the server, which does server-side fan-out to all group participants. The encrypted message will also contain $ID_{gr}$ and an IDMessage $ID_m$. The server adds the sender ID, a sender name and a timestamp to the message for the receivers.

When group members receive the message, they use the sender's public Signature key (received in the first message) to verify the signature, then they derive the Message key and update the Chain key in the same way as the sender did. Finally, they decrypt the message and delete the Message key.
As soon as all members in a group have received the message from the sender, the successful delivery is displayed by a double checkmark. Users can also send a new message while highlighting a reference to a previous message, similarly to direct messaging.
If the sender wants to send another message, she will derive a new Message Key from the updated Chain key. The earlier Message Key is deleted.

This design has a flaw: it does NOT offer perfect *future secrecy.* If an attacker compromises the Chain Key of a group member, then any future

message which is sent by that particular member will also get compromised. This happens because the group messaging algorithm uses only the symmtric ratchet: the Diffie-Hellman ratchet is skipped!
Whenever a group member leaves, all group participants clear their Sender Key and start the protocol again: this is a must because otherwise people who leave the group still have with them Sender Keys of every other member.

If an administrator inserts a new person in the group, all devices in the group chat must send their keys to this new group member, with the X3DH protocol. Administrators send group modifications to the server: if the server accepts the request, the trigger is implemented using a message denoted *group management message*, which has the format "Alice added Bob", "Alice removed Bob", and so on.
However, both Signal and WhatsApp fail to properly authenticate group management messages: they are not end-to-end encrypted or signed by the administrator. At least in theory, this makes it possible for a malicious server to silently add an infiltrator in the group chat. WhatsApp group chats were discussed the last year because a cryptographic analysis showed a potential flaw, where an infiltrator was added from the server, and then he could add more users to the chat or read old messages ([30], [31]). The WhatsApp server plays a significant role in group management, because it has in memory the list of the administrators in a group. When an administrator wants to add a member to the group, it sends a message to the server identifying the group and the member to add. The server checks if that user is an administrator, and then it sends a message to every member of the group indicating that they should add that user.
Since the group management messages are not signed by the administrator, a malicious WhatsApp server could add any user it wants into the group. We have to trust the WhatsApp server to not be corrupted, which seems a fair assumption, but it goes against the entire purpose of end-to-end encryption. The easiest fix here, which is still not available, is to let the administrators sign the group management messages.
One of the creators of Signal, Moxie Marlinspike, claimed that this attack is not possible: the attacker will not see any past messages to the group because the keys are protected from the Hash ratchet, which provides forward

secrecy. Moreover, all group members will see that the attacker has joined, and no one, not even the server, can suppress this message. The notify will always be present because it's an instruction for the key exchange.

Group chats (and, in a minor measure, direct messaging) are also victims of another attack. We have said how a double checkmark appears when the message sent is received from every other user in the group. This information can be faked by the WhatsApp server. A malicious server can manipulate the transcript between sender and receivers, dropping a message and sending a fake notification to the sender. The attack works in this way: when the sender writes a message in the group, the attacker intercepts a group message (for example, the one directed to Bob) and replies with a fake acknowledgment. In this way, the sender thinks every user has received the message, even though Bob never saw it! WhatsApp could fix this problem treating receipt messages like normal messages, end-to-end encrypting them. This would guarantee the authenticity of these messages.

## 5.2 SRTP and WhatsApp calls

Similarly to messages, WhatsApp video and voice calls are also end-to-end encrypted. To manage them, WhatsApp relies on Signal and on the Real-time Transport Protocol (RTP). RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio or video. The data transport is augmented by a Control protocol (RTCP) to allow the supervision of the data delivery to large networks, and to provide minimal control and identification functionality. A variant is the Secure Real-time Transport Protocol (SRTP), which can also provide confidentiality, message authentication, and replay protection. SRTP aims at a low bandwidth cost and a framework that allows for upgrades with new cryptographic transforms. SRTP provides some additional features over RTP:

- a "master key", to provide keying material for confidentiality and integrity protection. They are used in a key derivation function, which gives as output some session keys securely derived from the master key. By using an unique session key for every different session, each of them can be secured. In this way, if one session is compromised, all other sessions will still be safe;

- KDF can periodically refresh the session keys, so if one is compromised the amount of information the attacker can get is fixed;

- Salting keys could also be used to protect against pre-computation and time-memory attacks. To achieve this, the use of a master salt is recommended. This value must be random but can be public.

So, SRTP uses two types of keys: session keys and master keys. The session key is used directly in a cryptographic application, while a master key is a random bit string from which session keys are derived. Master keys must be random and kept secret. In the algorithm, the master key has usually 128 bits, while the master salt has 112 bits. WhatsApp doesn't follow this default because it uses AES256 instead of AES128. The rest of this Section will describe the algorithm with the default key lengths: just double every key length to get what WhatsApp does. Each SRTP stream requires the sender and receiver to maintain a cryptographic state information, named *cryptographic context*. There is also a corresponding of the RTCP protocol, named SRTCP. An accurate description of RTP and SRTP protocols is beyond the scope of the thesis; however, the interested reader can check [32], [33].
What is important for WhatsApp is that SRTP provides confidentiality by encrypting the RTP payload (i.e., the data transported by the RTP in a packet, for example audio samples). They encrypt data using AES with a particular mode, known as Segmented Integer Counter Mode (CTR), which handles traffic over an unreliable network with a possible loss of packets. A more known alternative, even if it is outdated, is the f8-mode, used for 3G mobile networks, which is a variation of Output FeedBack mode (OFB). Let's describe the two alternatives. In both modes, the encryption key $k_e$ has length 128 bits, while the salting key $k_s$ has length 112 bits and they are based on a block cipher, where the message is divided into blocks of 128 bits.

- **CTR mode**: nowadays, it is the default encryption algorithm mode. It consists in the encryption of successive integers. Each packet is encrypted with a distinct keystream segment, which is the concatenation of the 128 bits AES outputs. For each block, a new integer is added to an initalization vector IV. IV is 128 bits long and it is defined as

$$IV = (k_s * 2^{16}) \oplus (SSRC * 2^{64}) \oplus (i * 2^{16}),$$

where $SSRC$ is contained in the RTP header and $i$ is the SRTP packet index. Each term in the IV definition is padded with leading zeros to make the operation well defined.

Then, in formulas, the keystream is

$$E_{k_e}(IV)||E_{k_e}(IV+1 \mod 2^{128})||E_{k_e}(IV+2 \mod 2^{128})||\ldots$$

- **f8 mode**: in this mode, the Initialization Vector (IV) is determined in the following way:

$$IV = 0x00||M||PT||SEQ||TS||SSRC||ROC,$$

where $M$, $PT$, $SEQ$, $TS$ and $SSRC$ are informations contained in the RTP header, while $ROC$ comes from the cryptographic context. IV is not used directly, but it is the input of the block cipher (i.e., AES) under another key to produce an internal value: this is done to prevent an attacker from gaining known input/output pairs. In formulas,

$$IV' = E_{k_e \oplus m}(IV),$$

where $m$ is a constant string with length 128 bits, usually defined as

$$m = k_s||0x555\ldots5.$$

Finally, let $S(j)$ be a 128 bits string ($j$ is a counter) and let $L = \lceil \frac{N}{128} \rceil$, where $N$ is the required length of the message. Then,

$$S(0) = E_{k_e}(IV'),$$

$$S(j) = E_{k_e}(IV' \oplus j \oplus S(j-1)) \text{ if } j > 1.$$

The keystream is $S(0)||\ldots||S(L-1)$.

For both modes, HMAC-SHA1 is instead the algorithm used as the basis for ensuring message integrity. We call $M$ the data which will be authenticated. In the case of SRTP, $M$ should consist of some specific data which were not encrypted, concatenated with the *ROC*. We call $k_a$ the authentication key for HMAC. $k_a$ has usually 160 bits, while the tag has 80 bits. SHA1 is an outdated Hash function: it could be safer to swap it with a more modern one, like SHA2 or SHA3.

All three session keys are derived from a Key derivation algorithm: at least a step of the algorithm is required, never use directly the master key. More details of this algorithm can be found in [33].

WhatsApp handles calls over their app in this way:

- The initiator (Alice) uses the X3DH protocol to establish a session with the recipient (Bob), if it still doesn't exist;
- Alice generates a random 256 bits SRTP master secret key;
- Alice transmits an encrypted message $C$ to Bob that indicates an incoming call. $C$ also contains the SRTP master secret key;
- If Bob answers the call, a SRTP encrypted call starts.

WhatsApp allows people to have an end-to-end encrypted phone call through the app until a maximum of four people. One of pitfalls on the basic SRTP protocol is that it only encrypts the payload, leaving the header plain. We have already seen in Chapter 4 how a header usually contains important informations about the information flow. A more recent implementation of the algorithm allows the encryption of the header too.

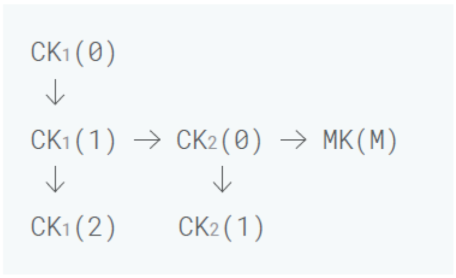
## 5.3 Statuses

WhatsApp status is a way to share our thought or point of view through the form of image, text or video. An user can choose to show their statuses to every contact or to a subset of contacts: from a cryptographic point of view, it's exactly the same of a group message, because we are essentially sharing a message into a finite group. For this reason, they are treated and encrypted exactly as group messages. The first status sent to the recipients (either all the contacts or a subset) follows the same sequence of steps as the first time a WhatsApp group member sends a message to a group. In the same way, new statuses sent to the same set of recipients follow the same sequence of steps as all subsequent messages to a group. The management of removed members from the subset of recipient is equal to the removal of an user from a group: the status sender clears their Sender Key and starts over. An user can be blocked from receiving a status either removing him from the list of valid receivers, or deleting his phone number/address. Statuses also suffer of every problematic we have discussed during the discussion of group chats, but in a minor measure: the lack of future secrecy doesn't provide security for the future statuses if the Chain Key is compromised, but the other two described attacks are not a real threat. In fact, given the nature of statuses, they are usually not used as a way to communicate, they are just a way to represent what we are doing in a certain moment or to share a memory. If an attacker compromises the device and is able to see statuses, probably she learns something about the hobbies of the victim, and that's all; if she doesn't allow Bob to receive the status, it is also usually not a big deal. Another WhatsApp parent company, Instagram, allows statuses (also known as stories) to be visible to every user of the application, if the sender wants to. It's clear that statuses content is in most cases of public domain.

## 5.4 Live location

Live location feature is real-time and dynamic. It allows users to share their location update in real-time. Users have the option to share their location with an individual as well as a group. They can also view the location of multiple people at the same time. The location will be shared depending on the selected time, it goes from 15 minutes until a maximum of 8 hours. An user can choose to stop the live location sharing in any moment. Like every other WhatsApp feature, Live Location supports end-to-end encryption. It is encrypted similarly to group messages and statuses.
The first live location sent follows the same sequence of steps as the first time a WhatsApp group member sends a message to a group. The management of new live locations after the first one is different instead. In fact, live locations require a high number of position updates, and for this reason receivers expect a large number of ratchet steps. The ratcheting algorithm described in the Signal protocol is too slow for this process. We need a new, faster, ratcheting algorithm. We denote $CK(i)$ the Chain Key at iteration $i$ and $MK(i)$ the Message Key at iteration $i$. In the classic double ratchet,

Chain Keys were one-dimensional: there was a single chain and the output of every step was a new Chain Key and a Message Key. If we did $N$ ratchet steps, we did $N$ computations, getting $CK(N-1)$ and $MK(N-1)$ (the iteration count starts from zero).
We can extend this algorithm, keeping track of more different chains of Chain Keys. Suppose we have two chains: denote them $CK_1(i)$ and $CK_2(i)$. Let $M$ be a positive integer constant. Suppose also that Message Keys are derived from $CK_2$ only. A receiver who needs to ratchet by a large amount can skip $M$ iterations at a time by ratcheting $CK_1$ and generating a new $CK_2$. In this way, a single iteration from $CK_1(0)$ to $CK_1(1)$ gives as output a Chain Key $CK_2(0)$ (it can be derived in many ways, the common choice is the use of a one-way function as we have seen during the Double Ratchet description), which then gives as output the Message Key $MK(M)$. $CK_2$ can be ratcheted too, until a maximum of $M$ times.
Differently from the $N$ steps we had to perform with just the Chain Key $CK_1$, now the number of steps is reduced up to a maximum of $\lceil \frac{N}{M} \rceil + M$ steps.



$$\begin{tikzcd}
CK_1(0) \arrow[d] & & \\
CK_1(1) \arrow[r] \arrow[d] & CK_2(0) \arrow[r] \arrow[d] & MK(M) \\
CK_1(2) & CK_2(1) &
\end{tikzcd}$$

Figure 5.1: Ratcheting with two Chain Keys

After a sender creates a new Message Key and encrypts a message with it, she has to delete all Chain Keys she generated to obtain it: taking as example the Figure 5.1, if the Message Key is $MK(M)$, then she has to delete $CK_1(0)$, $CK_1(1)$ and $CK_2(0)$. This is done to preserve forward secrecy.
This scheme can be generalized to $D$ dimensions, that is $D$ different Chain Keys. Every Chain Key $CK_l$ is derived from $CK_{l-1}$ through the use of a Hash function (with different inputs for every generated key), with the exception of $CK_1$, which is generated thanks to a random number generator.

Keys are 32 bytes long and initialized as

$$CK_1(0) = RNG(32) \text{ if } j = 1,$$

$$CK_j(0) = HMAC_{CK_{j-1}(0)}(\mathbf{j+1}) \text{ if } j > 1,$$

and ratcheted as

$$CK_j(i) = HMAC_{CK_j(i-1)}(\mathbf{j+1}),$$

where $\mathbf{n}$ denotes an array of bytes containing a single byte $n$.

$$\begin{tikzcd} \text{RNG} \arrow[r] & \text{CK}_1(0) \arrow[r] & \text{CK}_2(0) \arrow[r] & \ldots \arrow[r] & \text{CK}_D(0) \end{tikzcd}$$

Figure 5.2: Generation of $D$ Chain Keys

The Message Key is derived from a Chain Key in the same way it was derived during the Double Ratchet, so if we are at iteration $i$ of the chain $j$, the new Message Key is

$$MK = HMAC_{CK_j(i)}(\mathbf{1}).$$

The value $D$ must be a power of two less than or equal to the number of bits in the iteration counter. Common choices are $D = 1$, 2, 4, 8, 16 or 32. $D$ is accurately chosen from the implementer for the tradeoff between CPU and memory. Every different Chain Key has up to $M$ iterations. If a Chain Key reaches the $M-$th iteration, it can't be used anymore: there exists an algorithm to restore it to an useful Chain Key, if all $D$ are unusable.
Moving from one iteration count to another ratchets a single Chain Key a maximum of $M$ times. There is then an obvious upperbound to the number of ratcheting operations: $DM$.

## 5.5 Verifying keys

As already implied in the previous Chapter, every WhatsApp user can verify the keys of another user she is communicating with, to confirm nobody else is spying their conversations. This can be done by scanning a QR code or by comparing a 60 digit number.

The QR code contains a version (it refers to the number of modules contained in a symbol, see [21]), the user identifier for both parties (i.e., their phone number) and the 256 bits public identity key of both users. When Alice scans Bob's QR code, the public identity key into the code is compared to the Bob's public identity key memorized in the WhatsApp server.

The 60 digit number is instead obtained concatenating the two numeric fingerprints for each user's identity key. A *public key fingerprint* is a short sequence of bytes used to identify a longer public key. Fingerprints are created by applying a Hash function to a public key. Fingerprints are shorter than the keys they refer to and for this reason they can be used to simplify certain key management tasks. A WhatsApp user performs the following operations to obtain her public key fingerprint:

- The user identifier is converted into a byte sequence. The encoding must be deterministic, because the same fingerprint must be recreated in various occasions;
- the public identity key and the user identifier are iteratively hashed 5200 times. The Hash function used is SHA-512;
- The first 240 bits = 30 bytes of the output is kept, the rest is discarded;
- The 30 bytes are divided into six parts, each containing 5 bytes;
- Every part is converted into a sequence of 5 digits: it is considered as a big-endian (i.e. the most significant value is stored first) unsigned integer and it is reduced $\mod 100000$;
- the six parts are again concatenated, obtaining the 30 digit numeric fingerprint associated to the user identity key.

Fingerprint security completely relies on the Hash function. If an attacker finds an identity key whose fingerprint equals the fingerprint of one of her contacts, she could impersonate him. However, SHA-2 is considered secure, so this attack is not reliable. The numeric fingerprint is also known as *security code.* Numeric security codes have some advantages over other kinds of codes:

- they are easy to localize. WhatsApp has more than 1.5 billions of active users in the world, so using a code based on words was not a good option. Similarly, hexadecimal representations are not compatible with all alphabets, so they were discarded in the process too. Instead, all languages WhatsApp supports know what base 10 digits are. That's why the obvious choice was a numeric code;
- they are visually and audibly distinct;
- they are compact.

WhatsApp users may get a notify every time the security code for a contact changes.

It is likely that not every WhatsApp user verifies safety numbers or safety number changes. To prevent a potential pitfall, the WhatsApp server has no knowledge of which users have verified the security code. This feature is necessary, otherwise, if the WhatsApp server is corrupted, it could start a Man-In-The-Middle (MITM) attack directed to users who don't authenticate each other, and the attack would have success. Without that knowledge, a malicious server can always perform the attack, but it has to attack users randomly: if it attacks an user who performed the authentication step, it would be caught.

## 5.6 WhatsApp web

Whatsapp web is the WhatsApp application version which can be used in a web browser. It can be used in the Google Chrome Browser on our personal computers. Users can send and receive messages directly from the web browser. To guarantee end-to-end encryption, the web-client establishes a secure connection to the phone. The messages we send through WhatsApp Web are encrypted by the WebClient, decrypted by our phone, then encrypted again to satisfy the E2EE scheme and then sent to the recipient. In this way, the message can be read or created by both mobile and web clients. This means there is no E2EE between the two web clients. The mobile clients can read all the messages.
When opening WhatsApp Web for the first time, a Curve25519 key pair is generated and stored in the local storage of the browser. Then, the initial

setup is completed when the mobile client scans the QR code of the web client: this establishes trust between phone and browser installation, it implies that the user trusts the browser. After the secure connection between the web and mobile client is established, it can be used for all the communication between the two users. Basically the Web client is just a GUI for our phone client, securely connected over the web thanks to the QR code. WhatsApp Web requires our phone to be online during the entire session, otherwise end-to-end encryption would not be possible. The phone must also be connected because all messages (sent and received) are saved only in a database of our phone. However, to let WhatsApp web work without a lot of pending time, the browser caches some of these messages to be more responsive, so that if we change the active chat in the web application we don't have to wait for the request to go to our phone and then reply back with the information.

## 5.7 Transport security

The communication between WhatsApp clients and WhatsApp servers is layered within a separate encrypted channel. This framework uses Noise Pipes from the Noise Protocol Framework for long running interactive connections: it is based on Diffie-Hellman key agreement. It provides some additional features for the clients:

- extremely fast connection setup and resume;
- metadatas are encrypted, to keep them hidden from unauthorized observers: no information about the connecting user's identity is revealed;
- no client authentication secrets are stored on the server. The server only stores the client's public authentication key from the Curve25519 key pair. If the server's user database is ever compromised, no private authentication credentials will be revealed.

A Noise protocol begins with two parties exchanging *handshake messages* (i.e., they share the master key, WhatsApp does this with the X3DH algorithm). After the handshake, each party can use this shared key to send

encrypted transport messages - this is the role of the Double ratchet algorithm. The Noise framework supports handshakes where each party has a long-term static key pair and/or an ephemeral key pair. A Noise handshake is described by a simple language, consisting of tokens which are arranged into *message patterns*: they specify the DH public keys and the DH operations. Then, message patterns are arranged into *handshake patterns*, which specify the sequential exchange of messages that comprise a handshake.
A handshake pattern can be instantiated by DH functions, cipher functions, and Hash functions to give a Noise protocol. In particular, WhatsApp uses Curve25519, AES with Galois Counter Mode (GCM, a mode where data are not only encrypted, but also get a tag: it is similar to an AEAD algorithm) and SHA-256. More details on the Noise protocol can be found in [41].

### 5.7.1 The role of Metadata

When talking about privacy, there have been concerns related to user metadata. Metadata include informations about who we are communicating with, day and hour of the message/call, the length of the call, the dimension of a message. They also memorize when we are online, when we use the phone or when we are inside WhatsApp. WhatsApp server must know these basic informations, otherwise it can't know who is the sender, who is the recipient, and so on.
The metadata of the user are also encrypted when she is communicating with other parties. WhatsApp legal terms allow the application to store informations about metadata for delivered messages or calls. WhatsApp also asks the user to share her entire contact list with the app. In this moment, there is no option to not give informations about the contact list, or to add just some selected users. The implementation of this feature in the future will be useless, because WhatsApp already has the list of contacts of every user, so, even if a new member starts using the app, there is a big probability that her number and most of her contacts are already monitored by WhatsApp. So, they are encrypted during transit, but metadata are also stored on WhatsApp's servers. This is sufficient to create a profile and draw links between the communicating parties. These data are useful in certain situations, for example if the state or the government needs informations

about communications of a "dangerous user". However, if a hacker puts her hands in these data, she could easily trace a profile of every user.
Metadata may also be used from Facebook, the parent company of WhatsApp: they can be used to trace the user behaviour, which is used to target the user with some specific ads. In this context, group chats are problematic, because metadata contain informations about a small community which probably share an interest.
As of 2019, metadata are necessary to handle E2EE, but optimal forward and future secrecy properties have a price.

## 5.8 Backups

An user can decide to save her conversations on another application in order to keep old chats and at the same time to save memory space in the device. They can be saved everywhere, but one of the most obvious choices is Google Drive, since it is free and easy to use. In this case, users should have care, because WhatsApp explicitly tells to the user "media and messages you back up are not protected by WhatsApp end-to-end encryption while in Google Drive". While this is obvious, because data are now placed outside the application and so out of control, there are repercussions if the Google account is hacked, as the attacker would have access to our message contents.
It's also worth noting that E2EE guarantees the message we send can only be read by the recipient or vice versa, but it has no control over what the recipient does with our chat. If the recipient publishes our conversation on Google drive and then her account is compromised, our conversation with her is compromised too. The recipient could also do some screenshots of the conversations and put them somewhere else. For this reason, we have to trust people we are talking with, because message content is completely out of sender control or E2EE control after it is cached into the phone recipient.

# Chapter 6

# Conclusion

We have reviewed in detail the WhatsApp end-to-end encryption protocol, describing every cryptographic primitive used and how they are assembled to ensure security. The Signal protocol is open source and for this reason it was widely studied during the last years: it is safe to say that, as of 2019, it assures security for direct messaging in an asynchronous envinronment. WhatsApp code is not known, but its white paper specifies the technical details for most of its features (the only exception is WhatsApp web). While the direct messaging can be considered secure, we can't say the same for group messages. Some care should be taken when administrating groups to avoid potential infiltrations, but the real lack is the absence of future secrecy. Other features like live location are derived from Signal, but the protocol is reworked to keep care of the information flow. There are still no technical papers on the last added features, so users should have care with their use, even though the basic primitives used to implement them are all well known and considered secure.
Talking about cryptographic primitives, if one of them is discovered weak, the user should not worry: there are a lot of possible options, so if a day, for example, the discrete logarithm problem will be solved efficiently, elliptic curves will be forgotten and some different mathematical trick will be used to exchange keys, e.g. RSA. We have mentioned SHA-1 when describing the Secure Real-time Transport Protocol: this is a first example of outdated Hash function. Modern WhatsApp implementations should abandon it to use a more moden Hash function like SHA3, which has also a different structure with respect to SHA1 or SHA2.
We can safely claim end-to-end encryption is doing its job, and security can

only become better if people keep testing and fixing possible new bugs in the code. The only concern is the role of metadata: they are necessary to allow the transit of messages, but they are also dangerous if they go into the wrong hands. New developments of the protocol should fix some rules on their use, so that they are used only for transit and not for advertising. In any case, members who are using WhatsApp every day to keep in touch with other users can be quiet: no one is spying the content of our conversations, and the biggest security fall for an user remains the user himself.

# Bibliography

[1] Paar, Christof, Pelzl, Jan, Understanding Cryptography, A Textbook for Students and Practitioners, Springer-Verlag, 2010.

[2] Silverman, Joseph H., The Arithmetic of Elliptic Curves, Springer, 1986.

[3] Languasco, Alessandro, Zaccagnini, Alessandro, Manule di Crittografia: Teoria, Algoritmi e Protocolli, Hoepli, 2016.

[4] https://sefiks.com/2018/12/19/a-gentle-introduction-to-edwards-curves/

[5] René Schoof, Elliptic Curves Over Finite Fields and the Computation of Square Roots mod p

[6] https://www.techworld.com/security/best-secure-mobile-messaging-apps-3629914/

[7] http://fse.studenttheses.ub.rug.nl/10478/1/Marion_Dam_2012_WB_1.pdf

[8] Daniel J. Bernstein and Tanja Lange, Faster addition and doubling on elliptic curves.

[9] Mike Hamburg, Twisting Edwards curves with isogenies

[10] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters, Twisted Edwards Curves

[11] Daniel J. Bernstein and Tanja Lange, Montgomery curves and the Montgomery ladder.

[12] Craig Costello and Benjamin Smith, Montgomery curves and their arithmetic. The case of large characteristic fields.

[13] Marc Joye and Sung-Ming Yen, The Montgomery Powering Ladder.

[14] Daniel J. Bernstein, Curve25519: new Diffie-Hellman speed records.

[15] Daniel J. Bernstein, The first 10 years of Curve25519.

[16] Descriptions of SHA-256, SHA-384, and SHA-512.

[17] H. Krawczyk, M. Bellare, R. Canetti, HMAC: Keyed-Hashing for Message Authentication.

[18] H. Krawczyk, P. Eronen, HMAC-based Extract-and-Expand Key Derivation Function (HKDF).

[19] D. McGrew, An Interface and Algorithms for Authenticated Encryption.

[20] Katriel Cohn-Gordon, Cas Cremersy, Benjamin Dowlingz, Luke Garrattx, Douglas Stebila, A Formal Security Analysis of the Signal Messaging Protocol.

[21] Kinjal H. Pandya, Hiren J. Galiyawala, A Survey on QR Codes: in context of Research and Application.

[22] Wikipedia, "Legendre symbol — Wikipedia, The Free Encyclopedia." 2016. https://en.wikipedia.org/w/index.php?title=Legendre_symbol

[23] Wikipedia, https://en.wikipedia.org/wiki/Verifiable_random_function.

[24] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, Tanja Lange, Elligator: Elliptic-curve points indistinguishable from uniform random strings.

[25] Trevor Perrin, The XEdDSA and VXEdDSA Signature Schemes.

[26] Moxie Marlinspike, Trevor Perrin, The X3DH Key Agreement Protocol.

[27] Boaz Barak, Shai Halevi, A model and architecture for pseudo-random generation with applications to /dev/random.

[28] Trevor Perrin, Moxie Marlinspike, The Double Ratchet Algorithm.

[29] Trevor Perrin, Moxie Marlinspike, The Sesame Algorithm: Session Management for Asynchronous Message Encryption.

[30] https://gizmodo.com/whatsapp-security-flaw-lets-someone-secretly-add-member-1821947311

[31] https://blog.cryptographyengineering.com/2018/01/10/attack-of-the-week-group-messaging-in-whatsapp-and-signal/

[32] H. Schulzrinne, S. Casner, Track Packet, R. Frederick Blue, V. Jacobson, Columbia University, Packet Design, Blue Coat Systems Inc., RTP: A Transport Protocol for Real-Time Applications.

[33] M. Baugher, D. McGrew, Cisco Systems, Inc., M. Naslund, E. Carrara, K. Norrman, Ericsson Research, The Secure Real-time Transport Protocol (SRTP).

[34] Harold M. Edwards, A normal form for elliptic curves.

[35] Nidhi Rastogi, James Hendler, WhatsApp security and role of metadata in preserving privacy.

[36] Calvin Li, Daniel Sanchez, Sean Hua, WhatsApp Security Paper Analysis.

[37] Signal, Advanced cryptographic ratcheting.

[38] Paul Rösler, Christian Mainka, Jörg Schwenk, More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema.

[39] Signal, WhatsApp's Signal Protocol integration is now complete.

[40] Signal, There is no WhatsApp 'backdoor'.

[41] Trevor Perrin, The Noise Protocol Framework.

[42] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, Douglas Stebila, A Formal Security Analysis of the Signal Messaging Protocol.