Bachelor thesis
Computing Science

Radboud University

Analysis of key management in Matrix

Author:
Floris Hendriks
s4749294

First supervisor/assessor:
Prof. J.J.C. Daemen
j.daemen@cs.ru.nl

Second supervisor:
Dr. B.J.M. Mennink
b.mennink@cs.ru.nl

Second assessor:
Dr. C.E. Dobraunig
cdobraunig@cs.ru.nl

January 17, 2020

Abstract

This thesis presents an analysis of Matrix's key management. Matrix is an end-to-end encrypted and decentralised application layer protocol, developed by The Matrix.org Foundation C.I.C. The protocol is used, among other applications, to end-to-end encrypt messages in a decentralised chat system. To date, Matrix is not equipped with a clear and well-described overview on how keys enable end-to-end encryption in a decentralised network. This thesis therefore describes how keys in Matrix are established, used, stored, exchanged and verified. Moreover, the analysis also explores the limitations of Matrix's key management and potential improvements.

Contents

Chapter 1

Introduction

Nowadays, the internet is full of real-time centralised communication services that specialise in a particular functionality. Some individuals use a chat system because it has the best message encryption, while others use a different one because it has better integration with certain services. These broadly varying users' needs create a fragmented user base that is divided over a great number of chat systems. As a consequence, it is a challenge for people to keep track of which contacts use which chat systems. The comic of Figure 1.1 illustrates this issue:

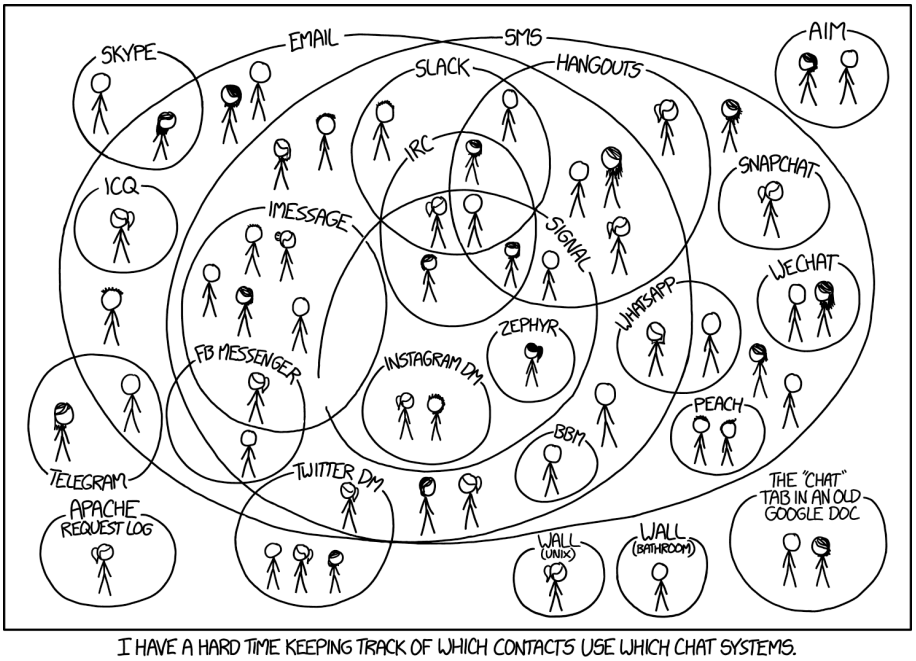

Figure 1.1: Schematic example of the defragmentation of chat services, taken from [38]

To mitigate this issue, "Matrix" is created by The Matrix.org Foundation C.I.C. [13]. Matrix is an end-to-end encrypted and decentralised application layer protocol, that enables unifying all communication of these chat services within one client using bridges and integration. Similarly to the Internet Relay Chat protocol [40], Matrix makes use of the client/server networking model. However, instead of the client being connected to a centralised server, every Matrix user has their clients connected to their own server. The server of a Matrix user is referred to as a "homeserver".

The homeserver can connect to an application server that functions as a bridge and form a "room" (see Figure 1.2). A room is a conceptual place where a user can send and receive messages. As an example, in Figure 1.2, Alice's client sent a message to her homeserver. Alice's homeserver can then, via the bridge, forward the message to a chat service (such as Whatsapp and Telegram) and vice versa. A drawback is that messages sent via the bridge are not end-to-end encrypted.

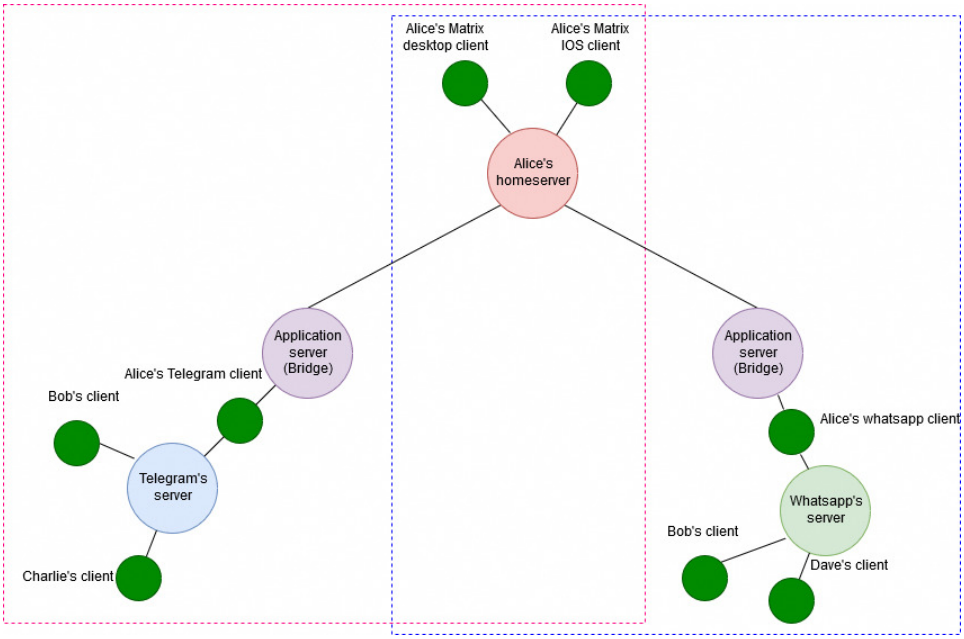

Figure 1.2: Schematic example of two rooms. One where Alice's client is able to exchange messages with Telegram, and the other with Whatsapp

Next to unifying communication services, The Matrix.org Foundation C.I.C. also developed a chat system based on Matrix. This is illustrated in Figure 1.3, where Alice, Bob and Charlie formed a room with their homeservers instead of bridges. Alice's client can send a message to her homeserver. The homeserver then stores the message and sends a

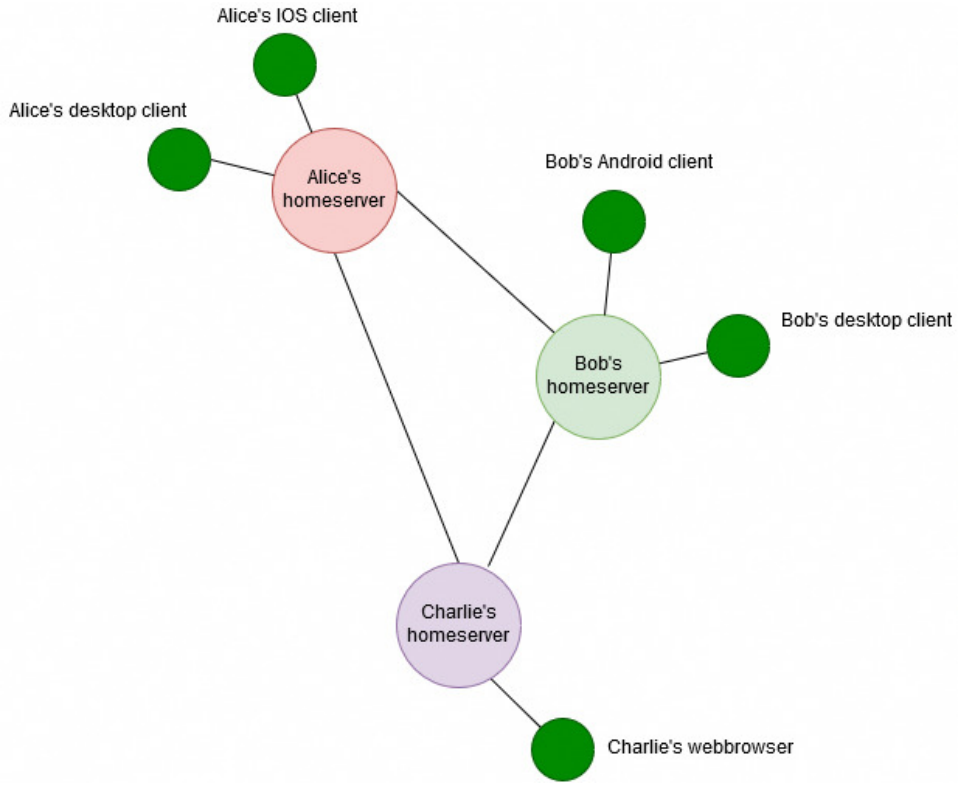

Figure 1.3: Schematic example of a room with Matrix users Alice, Bob and Charlie

copy to Bob's and Charlie's homeserver. Bob's and Charlie's clients are connected to their homeserver and can retrieve the message. Contrary to rooms with bridges, rooms formed between Matrix users are able to end-to-end encrypt their messages.

1.1 Research questions

End-to-end encryption in a decentralised network generally is a hard problem. Many projects tried to implement this, e.g. Origin Messaging [6], but failed due to overhead and scaling problems [19]. Matrix is one of the few projects that implemented end-to-end encryption in a decentralised network while avoiding overhead and solving scaling problems. This leads to the main research question:

- **How does Matrix manage keys to make end-to-end encryption of messages in a decentralised network possible?**

In other words, the main goal is to *understand* how keys are computed, verified, exchanged, stored and used. This is not as easy as it sounds; Matrix is a complex system and does not yet have a compact and self contained overview of the key management.

Furthermore, this paper will also look at the shortcomings of Matrix's key management:

- **What are the limitations of Matrix's key management and how can it be improved?**

1.2 Structure

The structure of this thesis is as follows. In Chapter 2 the data formats, cryptographic primitives and encoding used in Matrix is discussed. Chapter 3 gives a high-level description of Matrix. Chapter 4 examines how keys are managed in Matrix. Chapter 5 covers the related work. In Chapter 6, the architecture and key management of Matrix is discussed. Finally, Chapter 7 answers the research questions and recommendations are made for possible future work.

Chapter 2

Preliminaries

2.1 Data formats used in Matrix

The only data format that Matrix uses to exchange data, is JSON. JSON is a text syntax that makes it possible to create data objects that consist of key-value pairs [27]. As seen in the JSON example below, each object is encapsulated by curly braces. Within this object, there are keys and values that are separated by a colon (e.g. key:value). The data type of the key has to be a string. A value can be one of either an object, number, array, true, false, string or null. Each key-value pair is separated by a comma. Matrix chose this format because it is easy to parse and human-readable [3].

Example of a JSON object

```
{
        "name": "Example",
        "author": {
                "name": "John Doe",
                "email": "JohnDoe@example.com",
                "contact": [
                        {
                                "location": "office",
                                "number": 5729294
                        },
                        {
                                "location": "home",
                                "number": 3451483
                        }
                ]
        }
}
```

2.2 Security principles

Security principles are guidelines to make a system secure. These are discussed throughout the thesis when exploring the key management of Matrix.

2.2.1 Forward secrecy

Forward secrecy ensures that when a key is compromised, previous established keys are not compromised.

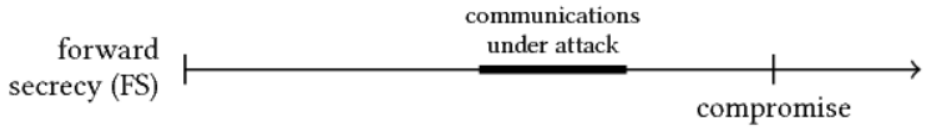

Figure 2.1: Forward secrecy, taken from [14]

2.2.2 Backward secrecy

Backward secrecy or post-compromise security ensures that if a key is compromised, the keys computed afterwards remain uncompromised.

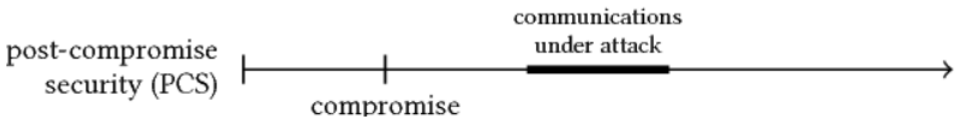

Figure 2.2: Backward secrecy, taken from [14]

2.2.3 Deniability

Deniability is often considered when designing key management. In some cases, the construction and usage of a key to encrypt the message can prove that participants were involved in a conversation. If key management has the property "deniability", participants are able to deny contents of their communication or the fact that they communicated.

2.2.4 Confidentiality

Confidentiality means that a sent message can only be read by the intended receiver.

2.2.5 Integrity

Integrity guarantees that a message has not been altered by an adversary throughout the transmission.

2.2.6 Authentication

Authentication guarantees that the receiver of the message can verify the sender.

2.3 Cryptographic primitives used in Matrix

Cryptographic primitives are low-level algorithms. These are the building blocks of cryptographic protocols. Therefore, to understand the key management in Matrix, the individual components need to be understood first.

2.3.1 Cryptographic hash functions

A hash function takes some arbitrary length string as input, and maps it to a bit string of a fixed size. A cryptographic hash function is a hash function that complies with the following properties [17]:

- Collision resistance, it is difficult to find inputs $x_1 \neq x_2$ such that $H(x_1) = H(x_2)$.
- Pre-image resistance, given a hash value y, it is infeasible to find any input x such that $H(x) = y$.
- Second pre-image resistance, given a value x_1, it is computationally hard to find a different input x_2 such that $H(x_1) = H(x_2)$.

A cryptographic hash function should be a one-way function. In other words, when the output is computed from the input, it should be infeasible to invert it. An example of a cryptographic hash function is SHA-256 [21], which is used by Matrix.

2.3.2 HMAC

An example where Matrix uses SHA-256, is when computing the Hash-based Message Authentication Code (HMAC). Generally, HMAC is used to verify data integrity and authenticity of a message. However, it is used by Matrix to advance keys. HMAC takes as input K, the key that needs to be advanced and arbitrary input m. The key is XORed with

"opad" and "ipad" before it is hashed. Opad and ipad are a block-sized repeated value. Following Bellare et al. [32]:

$$HMAC(K, m) = H((K' \oplus \text{opad}) \parallel H((K' \oplus \text{ipad}) \parallel m))$$

$$K' = \begin{cases} H(K), & \text{if } K \text{ is larger than the block size} \\ K, & \text{otherwise.} \end{cases}$$

2.3.3 HKDF

HKDF [31] is the abbreviation for Hash-based Key Derivation Function. It is designed to take some source key and a salt as input, in order to convert it to one or multiple cryptographic strong secret keys. HKDF consists of two stages:

1. Extract,
2. Expand.

Extract

Using HMAC that has a salt and some input key material as input, a new fixed-length key K is "extracted". The length is determined by the hash that is used:

$$K = HMAC(\text{salt}, \text{key material}).$$

Expand

To derive new keys from the extracted key, the key is "expanded" with a KDF. It takes as input the extracted key K, a context string CS, a cryptographic hash function H, and an integer L which implies the desired length. The output is the desired key length:

$$KDF(K, CS, L) = H\Big(K, (CS \parallel 0)\Big) \parallel H\Big(K, (CS \parallel 2)\Big) \parallel \cdots \parallel H\Big(K, (CS \parallel L)\Big).$$

2.3.4 Cryptographic ratchets

Ratchets are chains of cryptographic primitives. A part or the whole output of a cryptographic primitive is the input of another cryptographic primitive (see Figure 2.3). Matrix utilizes this with HMAC or HKDF as cryptographic primitive to indefinitely advance keys, while guaranteeing forward secrecy.

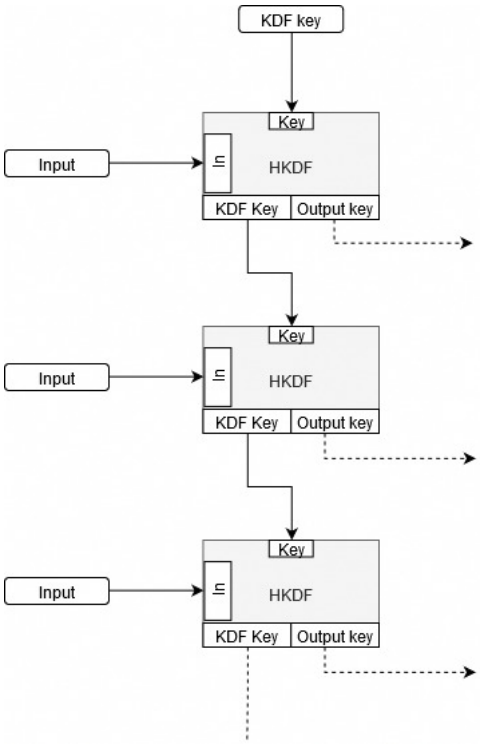

Figure 2.3: Example of a cryptographic ratchet

2.3.5 Curve25519

Curve25519 [9] is an elliptic curve with defined domain parameters. An elliptic curve E over $\mathbb{R}$ is defined as a set of pairs (x, y) satisfying:

$$y^2 = x^3 + ax + b.$$

Elliptic curves are non-singular, meaning that the curve has no cusps or self-intersections [30]. This is useful because cryptographic operations (such as point addition and multiplication, see Figure 2.4) on the curve require that every point on the curve has a unique derivative. A function that is self-intersecting or has a cusp does not have a unique derivation at that specific point. By using these cryptographic operations, Curve25519 is able to generate key pairs and, combined with Diffie-Hellman shared secret keys (ECDH). Curve25519 defines the following domain parameters [9]:

- the prime number $p = 2^{255} - 19$,
- A finite field, $\mathbb{F}_p$,
- Elliptic curve E, $y^2 = x^3 + 486662x^2 + x$,

- The base point and generator $G \in E(\mathbb{F}_p)$ with $n = ord(G)$.

Generating key pairs with Curve25519

The private key is created by selecting a random integer $d \in \{1, \ldots, n-1\}$. By scalar multiplying (see Figure 2.4) the base point G with d, a new point is computed [20]:

$$d \cdot G = Q.$$

The x-coordinate of point Q is used as the public key.

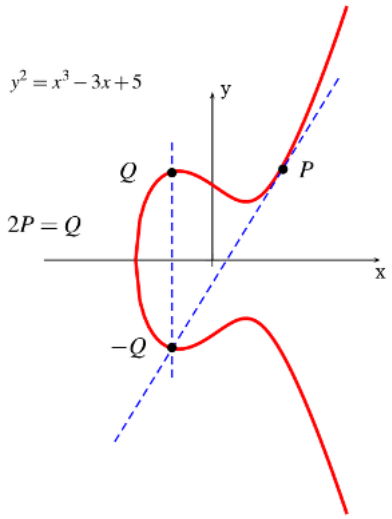

Figure 2.4: Example of scalar multiplication in an elliptic curve, taken from [16]

Elliptic curve Diffie-Hellman

Assume Alice and Bob want to compute a shared key using ECDH. They both create a key pair as described above. Let Alice key pair be (d_A, Q_A) and Bob's key pair be (d_B, Q_B). Alice then sends Q_A to Bob and Bob sends Q_B to Alice. Alice computes the shared secret S by multiplying her private key with Bob's public key: $d_A \cdot Q_B = S$. Bob computes the same shared secret S by multiplying his private key with Alice's public key:

$$d_B \cdot Q_A = S.$$

Their shared secret key S is the same since:

$$d_A \cdot Q_B = d_A \cdot d_B \cdot G = d_B \cdot Q_A.$$

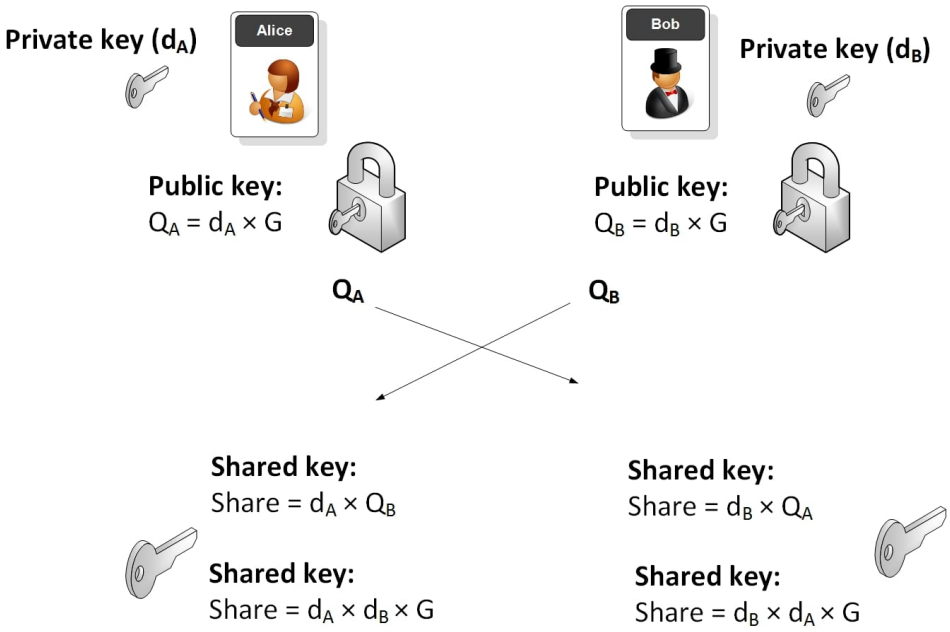

Figure 2.5: Schematic example of a computed shared secret between Alice and Bob, taken from [10]

2.3.6 EdDSA and Ed25519

Edwards-curve Digital Signature Algorithm (EdDSA) is a public-key signature scheme, that defines parameters and operations to securely sign data. The values of those parameters have to be chosen carefully for a secure and efficient signature [29]. Bernstein, Duif, Lange, Schwabe and Yang therefore researched which parameters result in a fast and secure signature [8] and presented Ed25519, an instantiation of EdDSA. Matrix uses Ed25519 to sign keys.

2.3.7 Triple Diffie-Hellman

To compute a shared key between two parties, Matrix uses triple Diffie-Hellman over Curve25519. Assume Alice and Bob want to establish a shared key using triple Diffie-Hellman [34]. First, they both have to create an identity key pair I and a one-time key pair O using Curve25519. They then send their public keys to each other. They are now able to compute the shared key S, which consists of three different Curve25519 computations. First, Curve25519 is used with as input I_A and O_B, the second time with O_A and I_B and the third time with O_A and O_B. All

these three secret shared keys are then concatenated into key S.

$$S = ECDH\left(I_A, O_B\right) \parallel ECDH\left(O_A, I_B\right) \parallel ECDH\left(O_A, O_B\right)$$

Where each party uses its own private key and the remote party's public key.

A claimed advantage of triple Diffie-Hellman, when comparing it to regular Diffie-Hellman, is that it guarantees deniability. Neither of the three computations takes as input both the identity key of Alice and the identity key of Bob. This is to ensure that if, for example, the identity key of Alice is compromised, Bob can deny that they established that key. Another advantage is that the one-time key pairs can be deleted after computing the shared secret, which makes it harder for an adversary to compromise the shared secret.

2.3.8 AES in CBC mode

Once a proper cryptographic key has been constructed using the primitives above, it still needs to securely encrypt the plaintext. To do this, Matrix uses the Advanced Encryption Standard (AES) [15] in Cipher Block Chain (CBC) mode [18]. Since AES is a block cipher which encrypts 16 bytes at a time, the plaintext m is split up into blocks of 16 bytes:

$$m_0 \parallel m_1 \parallel \cdots \parallel m_{i-1} = m.$$

If the length of m is not a multiple 128 bits, the plaintext needs to be padded. The type of padding that Matrix uses is PKCS#7 [26], which adds padding in whole bytes. The value of the padded byte depends on the number of bytes that are needed to be added:

$$\begin{aligned}
&01, \text{if 1 byte needs to be padded.}\\
&02\ 02, \text{if 2 bytes need to be padded.}\\
&03\ 03\ 03, \text{if 3 bytes need to be padded.}\\
&\vdots\\
&k\ k \ldots k\ k, \text{if k bytes need to be padded.}
\end{aligned}$$

As an example, take a 9 byte message string with a block size of 16 bytes:

$$m = AA\ AA\ AA\ AA\ AA\ AA\ AA\ AA\ AA.$$

5 bytes need to be padded to make the message 16 bytes long. Therefore, 5 bytes with the value 05 are padded to the string:

$$m = AA\ AA\ AA\ AA\ AA\ AA\ AA\ AA\ AA\ 05\ 05\ 05\ 05\ 05.$$

After padding is appended to the plaintext (if necessary), the plaintext is split up into blocks of 16 bytes. The first plaintext block m_0 is XOR-ed with a random 16 byte Initialization Vector (IV) which results into a bit string I_0:

$$I_0 = m_0 \oplus IV.$$

The AES block cipher then symmetrically encrypts the byte string I_0 with a key k into a block of ciphertext c_0:

$$c_0 = AES(I_0, k).$$

To compute the i-th input byte string it is not XOR-ed with the IV but with the previous ciphertext:

$$I_i = m_i \oplus c_{i-1}.$$

Computing the i-th block of ciphertext remains the same:

$$c_i = AES(I_i, k).$$

The blocks of ciphertext are then concatenated back to one long ciphertext so it can be sent via an unsecure channel to the intended recipients.

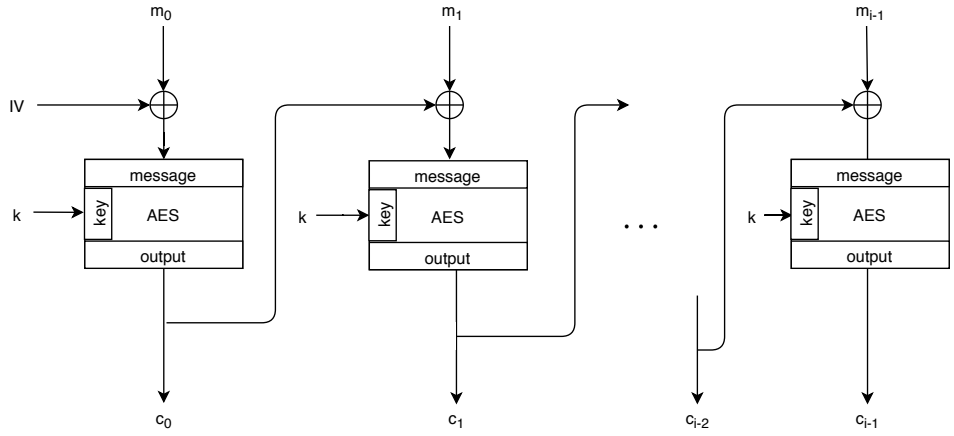

Figure 2.6: Schematic example of AES in CBC mode

2.4 Encoding

To encode keys and ciphertext, Matrix uses unpadded Base64. The bit strings are split into blocks of 6 bits, and then converted to decimal. Next, each decimal is converted to a character using the Base64 table:

Value	Character	Value	Character	Value	Character	Value	Character
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

Figure 2.7: Base64 table, taken from [5]

Chapter 3

A high-level description of Matrix

3.1 Architecture

3.1.1 Clients and servers

As previously mentioned, if a person wants to make use of Matrix, he/she needs a client and a homeserver. The user is free to choose which client. The user can prefer Riot, the client developed by The Matrix.org Foundation C.I.C., but also third-party developed clients such as Quaternion, Spectral and nheko Reborn [1]. The user needs to connect its clients with its homeserver. A server is not able to immediately communicate with a Matrix client. The server first needs to install "Synapse". Synapse is developed by The Matrix.org Foundation C.I.C. so that the Matrix protocol can refer to it as a homeserver. In other words, Synapse ensures that data received and sent on the server, is handled correctly according to the Matrix protocol. If a user does not own a server, the user is able to make use of Matrix's servers.

3.1.2 Users and device keys

To send messages through the client, a user account and a so-called "device" are needed. As seen in Figure 3.1, each user is identified by: @username:domain, where domain is the current homeserver that the user uses.

If a user logs into a client for the first time on a client supported platform (multiple web browsers, Linux, Windows, iOS, Android, etc.) it is registered as a "new device". For this new device a set of keys is generated.

Each device has a number of long-term keys and one-time keys. One of the long-term keys consist of an Ed25519 fingerprint key pair, which is used for signatures and key verification. The other long-term key pair is a Curve25519 identity key pair used for shared secret key establishment [2]. The one-time key pairs are made by using Curve25519, which are also used for shared secret key establishment. The public part of the key pairs is published to the homeserver using the client-server API [45]. The private part of the key pairs remains on the client. These keys are later on used to end-to-end encrypt (see Section 4.2).

3.2 Establishing a room

After the user has set up a client on a platform and connected it to its homeserver the user can join or create rooms. Each room is identified by a unique room id, of the form !string:domain. The homeserver of a Matrix user keeps track of which rooms the user participates in and who the members are of that room. If a user sends a message in the room, the user's client sends the message along with the room id to the homeserver. The homeserver checks who participates in the room and forwards the message to the homeservers of those members. The client of the other members can retrieve the message from their homeserver. As an example, consider Figure 3.1. If Alice sends a message in room !fwggEEwdw:matrix.org, the homeserver examines the room id and sends it only to the corresponding recipients which is, in this case, Charly's and Dave's homeservers. If Bob wants to join room !fwggEEwdw:matrix.org, Alice, Charly or Dave can send an invitation to the homeserver of Bob via their own homeserver. Bob's clients receive the invitation from his homeserver. If Bob accepts the invitation, he sends data to his homeserver that he accepts. The homeserver of Bob then sends data of Bob's approval to the other homeservers. Alice, Charly's and Dave's homeservers read the data and know that Bob accepted their invitation. Bob is added as a member by the homeservers to the corresponding room id.

If Alice, for example, wants to have a one-on-one conversation with Bob, she can create a new room. One of Alice's clients generates a new room id and sends it to her homeserver. The homeserver now associates the room only with Alice. She can invite Bob to her newly established room by sending data about the room from her homeserver to Bob's homeserver. Similarly to the above, Bob can accept the invitation and Alice's homeserver and Bob's homeserver correspond the new room id with the

members Alice and Bob. If Alice sends a message in the room, her client sends the message along with the room id to her homeserver. Her homeserver then forwards the message to Bob's homeserver. Bob can retrieve the message from the server by using his client.

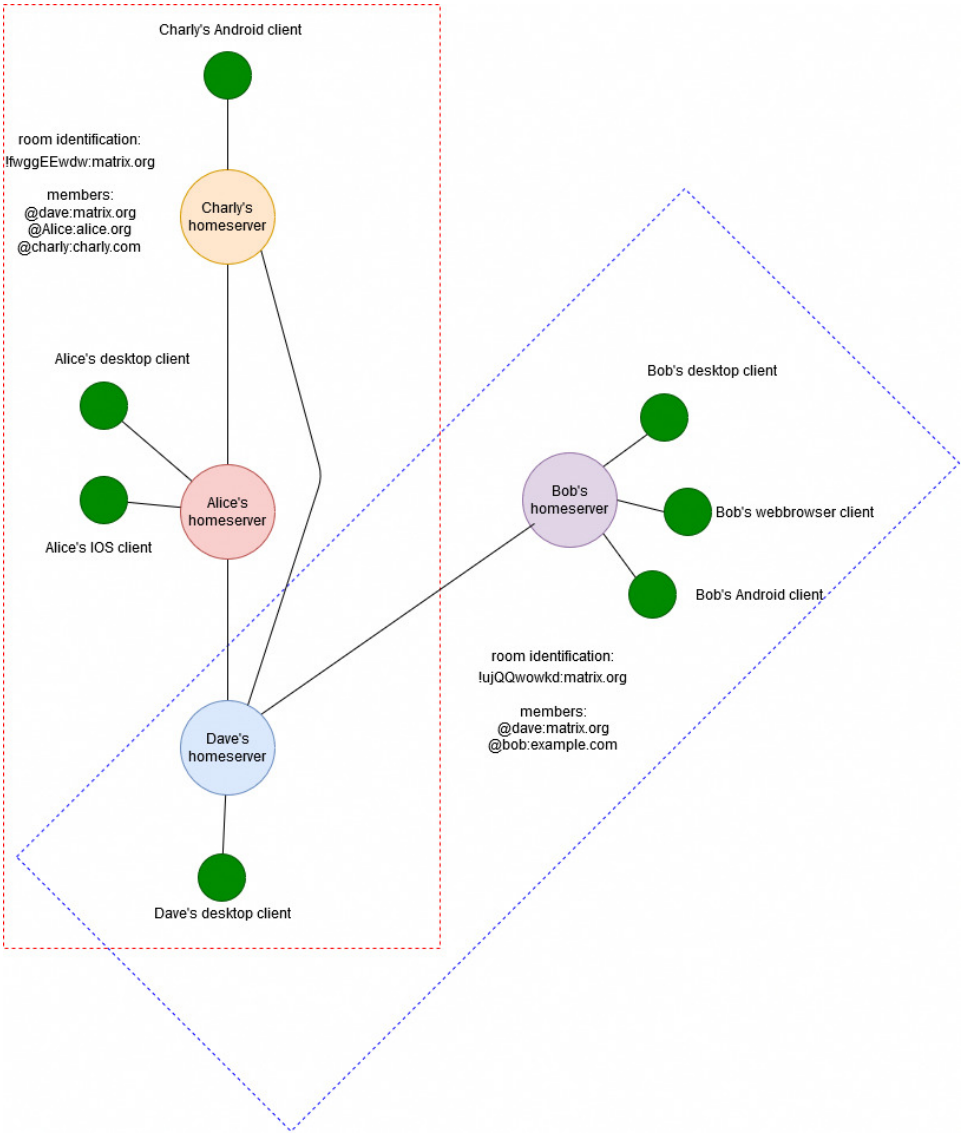

Figure 3.1: A schematic example of two rooms. One with Alice, Charly and Dave and the other with Dave and Bob

3.3 Message communication flow

To go into more detail of how messages are exchanged between users, we continue with the example where Alice and Bob establish a room. Assume Alice wants to send a message to Bob. Alice first sends her message to her homeserver via the HTTP PUT protocol. Alice's homeserver stores the message and "federates" (see Section 3.4.2) the message to Bob's homeserver. Bob's homeserver receives and stores the message. By using a long-polling HTTP GET request [43], Bob is able to retrieve Alice's message [3]. Say Alice wants to read the message she sent on her desktop on a new device, her iPhone. Alice logs into her mobile client, and the client sends a GET sync request to her homeserver. The server processes this request and if it is approved, the server sends the room's past messages and state events. See Figure 3.2.

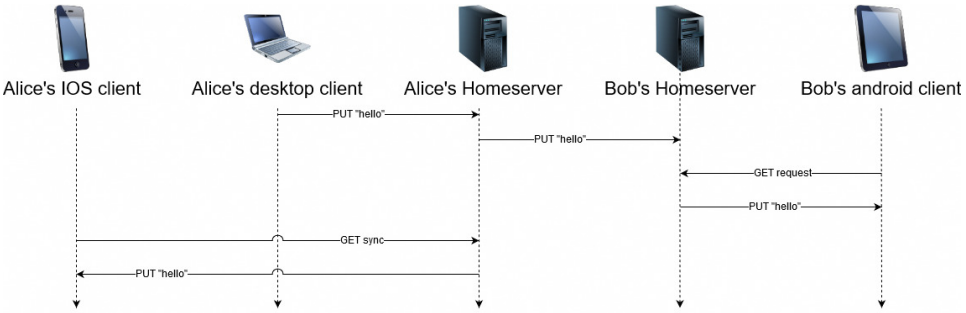

Figure 3.2: Schematic architecture of a sent message and a sync request

3.4 Communication data

3.4.1 Events

As seen in 3.2, a message from the client is sent to the homeserver among other data including the room id. This is needed by the homeserver to be able to forward the message to the other members of the room. Matrix formats this data as a JSON object, so that the homeserver can easily parse the room id, the message and other necessary data. Matrix refers these JSON objects that are exchanged between client and homeserver, and also between homeservers as an "event". Matrix splits these events into two types:

- Message event, data that users use to converse with one another. Some examples are a chat message or a VoIP call.

- State event, data that creates a room or updates information about a room. This information is needed by the homeserver to authenticate events received from a client, and to forward events to the members of the room. Examples are: an invite for a room, a change to the room's name or a banned member.

Each event has a content and a type field. The type field determines which extra fields are required. Take for example Figure 3.3 which is a message event in an end-to-end encrypted room. The type of the room is m.room.encrypted. The type of the event is a message in an end-to-end encrypted room. This event requires to specify:

- An algorithm, the algorithm that is used to encrypt the message which is in this case Megolm (explained in Section 4.2.3).
- A ciphertext, an encrypted message that the sender wants to send to the other members of the room. How this is constructed is explained in detail in Section 4.2.
- The device_id, this specifies which device the sender used.
- The sender_key, the public identity key of the device.
- The session_id, the session_id specifies which Megolm session has been used.
- The origin_server_ts, the timestamp of when the event was sent by the sender's homeserver.
- The sender, which is the Matrix identification of the sender.
- Unsigned, this field is not going to be signed by the homeserver (more information about signing in Section 3.4.2) since it needs to be edited by other homeservers. An example here is the age parameter, this tells in milliseconds how long it took between sending and receiving the event which will change in transit.
- An event_id, the identification of the event.
- The room_id, the specified room this message has to be sent to.

Figure 3.3: Example of a message event in an end-to-end encrypted room

When a client changes the room name, a state event is sent to the homeserver. Take for example Figure 3.4, which changes the room name from “Test conversation” to “Hello world!”. This event has a lot of fields in common with the message event. However, it does require some different fields:

- name, the room’s new name.
- state_key, tells the server to interpret this event as a state event.
- replace_state, the previous state that is being replaced.
- prev_content, the previous room’s name.
- prev_sender, the Matrix identification of the sender of the previous event.

Note that the content of state events is never encrypted, not even in an end-to-end encrypted room. This is because the homeservers need to know about the current state of the room, so it is up to date with the current members, name changes, ban list, etc.

Figure 3.4: Example of a state event in an end-to-end encrypted room

3.4.2 Verifying events

Whenever a homeserver receives an event from the client, it is not immediately distributed to the other homeservers in the room. To ensure authenticity and integrity of the event when sharing the event, the homeserver hashes and signs the event. First, the content of the event is hashed using SHA-256 and appended to the event. After the hash, the whole event (excluding the "unsigned" field) is signed by the homeserver using its private Ed25519 key. The signature is then appended to the event. It is then sent to the other homeservers in the room. This process of synchronizing events with other homeservers is called "federation". In this way every homeserver in the room is up to date with the current state of the room, and clients are able to retrieve new messages from the homeservers.

When a homeserver receives an event from a remote server, it verifies the event before a client is allowed to retrieve the event. If the event fails one of the following checks below, the homeserver determines whether it is dropped, redacted or rejected [42]:

- If it is not a valid JSON event, otherwise it is dropped.
- The signature check, otherwise it is dropped
- The hash check, otherwise it is redacted.

- Authorisation rules based on the event's auth events, otherwise it is rejected.
- Authorisation rules based on the state at the event, otherwise it is rejected.
- Rules based on the current state of the room, otherwise it is rejected.

If the event passes the checks above, the signature and hash are stripped from the event. The homeserver then waits until the client requests the event.

Authorisation rules

Authorisation rules are the rules of the room that the participants of the room can define. The creator of the room is the admin and he/she can change the power levels of other participants. Depending on the power level, some actions can be taken. Examples of these actions are redacting events of other members, or banning them. Homeservers, therefore, check if the action of the event from the member complies with their power level.

Redacting

Once an event has been federated to the other servers in the room, it is impossible to delete that event. With redacting, all the JSON keys that are not required by the protocol are stripped off. In that way, the client can receive the redacted event but not see the contents.

Figure 3.5: Example of a redacted message event in an encrypted room

Rejecting

If a user wants to invite another user but is not allowed to because only the admin is authorized to do that, the state event is rejected by participating servers in the room. Unlike a redacted event, the client is not able to retrieve a rejected event.

Chapter 4

Key management in Matrix

4.1 Verifying clients

Assume Alice and Bob want to have a private encrypted conversation with each other. Before establishing shared secret keys between Alice's and Bob's clients, the users have the option to verify each other clients to prevent impersonation by an adversary. Manually comparing the Ed25519 fingerprint keys is possible but can lead to human errors since these are 43 character long Base64 strings. To make it less prone to these errors, Matrix implemented Short Authentication String (SAS) verification [2]. The verification protocol only verifies the fingerprint keys, because if the fingerprint key is verified, the identity key can be verified. This is achieved when a client sends the device keys to another client. The client signs the identity key with the fingerprint key so that the receiver can verify the identity key. The verification protocol goes as follows:

1. Alice and Bob meet by using an out-of-band channel, this can be done by meeting physically or a VoIP (voice over IP) call.
2. They communicate with each other which devices they want to verify.
3. Alice's client starts by sending a m.key.verification.start event. This event contains information about Alice's device which key agreement protocols (most of the time Curve25519), message authentication codes (most of the time HKDF SHA-256), hash algorithm (often SHA-256) and SAS methods (Decimal or Emoji) it understands.
4. Bob's client receives the event and chooses which key agreement protocol, hash algorithm and SAS method are going to be used (see Section 4.1.1).

5. Bob's client generates a key pair $(K_B^{private}, K_B^{public})$ using Curve25519. By using the chosen hash algorithm, Bob hashes K_B^{public}, which results into H_B.

6. Bob's client creates a new m.key.verification.accept event. This event contains Bob's chosen algorithm, protocol and method. Bob also adds the hash H_B so that later on, Alice can verify that Bob has sent the correct K_B^{public} and not a different key sent by a man-in-the-middle. Bob's client sends the event to Alice's device.

7. Now Alice's client generates a key pair $(K_A^{private}, K_A^{public})$ using Curve-25519. Alice device sends to Bob's device a m.key.verification.key event. This event only contains K_A^{public}.

8. Bob's client receives the event with K_A^{public} and replies with another m.key.verification.key event but this time with his own K_B^{public}.

9. Alice's client receives the event with K_B^{public} and verifies if the hash of K_B^{public} matches with H_B.

10. Using their private key $K^{private}$ and the other party's public key, Alice and Bob are now able to compute their shared secret key S using ECDH.

11. Using their S and their SAS method, the SAS is computed (see Section 4.1.1).

12. If the SAS of Alice and Bob are similar, they both calculate a HMAC with the public part of their device's fingerprint key and S as input. Then Alice and Bob create a m.key.verification.mac event. This event contains the computed HMAC and is sent to the other party.

13. The receiving party checks if the HMAC is the same as the HMAC with as input the remote device's key and S. If they are the same then the devices are verified.

The event exchange of verifying the devices between Alice and Bob is as follows:

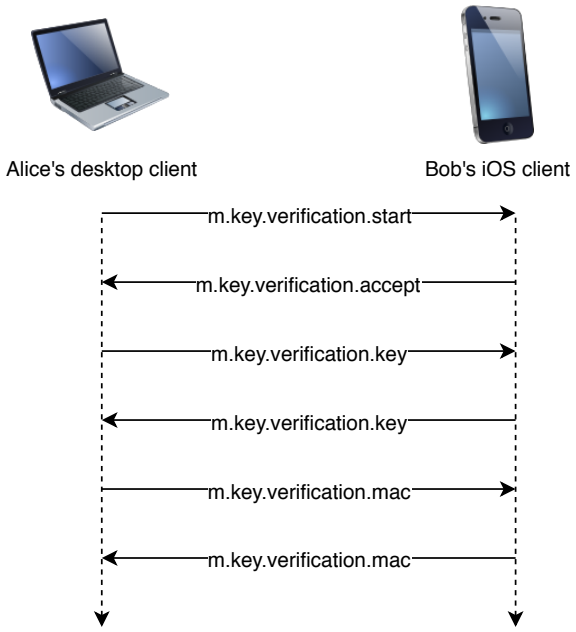

Figure 4.1: Schematic event exchange of Alice and Bob devices when verifying (Alice's and bob's homeservers only forward the messages and are therefore abstracted from this diagram)

4.1.1 Computing the SAS

There are two methods to generate a SAS, by using decimals or emojis. Older devices that do not support emoticons use decimals for the verification. Both use HKDF with as key material the shared secret S. The input parameter of the HKDF is not a salt, but a string based on the Matrix and device id of the start and accept events. The number of bytes the HKDF needs to output depends on the SAS method.

SAS method: decimal

This method takes 5 bytes from the output of the HKDF identified by: B_0, B_1, B_2, B_3 and B_4. With these 5 bytes, 3-bit strings of 13 bits are computed from arbitrary chosen bitwise operations:

- first bit string: $(B_0 \ll 5 \mid B_1 \gg 3) + 1000$.
- second bit string: $\Big((B_3 \,\&\, 0x3F) \ll 7 \mid B_4 \gg 1\Big) + 1000$.

- third bit string: $\Big((B_3 \,\&\, 0x3F) \ll 7 \mid B_4 \gg 1\Big) + 1000.$

One thousand is added to each bit string so that when converting the bits to decimal no padding is needed. These three decimal numbers are compared with the numbers of the other device. If they are similar, the above verification protocol continues.

SAS method: emoji

This method takes 6 bytes from output of the HKDF. The first 42 bits are split into 7 groups of 6 bits. Each group is converted to an emoji. These 7 emojis are then compared to the emojis of the other device. When corresponding, the above verification protocol continues.

4.2 End-to-End message encryption and authentication

Encrypting and authenticating a message makes use of two different cryptographic ratchets.

- Olm ratchet [33] (heavily based on the double ratchet algorithm [35]), see Section 4.2.1.
- Megolm ratchet [25], see Section 4.2.3.

First, an Olm ratchet is established between two users to create a secure channel. If that succeeds and for example Alice wants to send a message to Bob, Alice constructs a Megolm session (single ratchet). Alice then encrypts the Megolm session with the produced Olm key and sends the encrypted session to Bob. Bob receives Alice's Megolm session and decrypts it. Alice encrypts her message with the Megolm key and sends the encrypted message to Bob. Since Bob has Alice's Megolm ratchet, he is able to retrieve the key that Alice used to encrypt the message. Bob decrypts the message and can read the content.

4.2.1 The Olm algorithm (double ratchet algorithm)

If Alice and Bob want to talk to each other via Matrix they use their generated device keys. These are the identity key pairs, I_A and I_B, and the one-time key pairs O_A and O_B. The secret shared key S is computed by triple Diffie-Hellman (see Section 2.3.7).

The 256 bit root key, R_0 and the first 256 bit chain key is computed using

HKDF with SHA-256 as the hash function. It takes as input: 32 bytes of zeroes as salt, “OLM_ROOT” as context string, key S and wanted output length in bytes, which is 64.

$$R_0 \parallel C_{0,0} = HKDF\left(0,\, S,\, \text{“OLM\_ROOT”},\, 64\right)$$

Advancing a root key makes use of the HKDF as well. It takes as input: the previous root key R_{i-1} as a value for the salt, a shared secret generated by Curve25519 taking as input the current ratchet key T_i and the previous T_{i-1}, the context string “OLM_RATCHET” and the wanted output length of 64 bytes.

$$R_i \parallel C_{i,0} = HKDF\left(R_{i-1},\, ECDH\left(T_{i-1},\, T_i\right),\, \text{“OLM\_RATCHET”},\, 64\right)$$

A message key and an advanced chain key can be computed using HMAC and the current chain key.

Creating a message key: $M_{i,j} = HMAC\left(C_{i,j},\, \text{“\textbackslash x01”}\right).$
Advancing the chain key: $C_{i,j} = HMAC(C_{i,j-1}, \text{“\textbackslash x02”}).$

Where i is the number of Diffie-Hellman ratchet iterations and j the number of times the sending or receiving ratchet iterated.

Using the message key, an AES_KEY, IV and HMAC_KEY can be generated using HKDF:

$$AES\_KEY_{i,j} \parallel HMAC\_KEY_{i,j} \parallel AES\_IV_{i,j} = HKDF\left(0,\, M_{i,j}, \text{“OLM\_KEYS”},\, 80\right).$$

With the AES_KEY, IV and HMAC_KEY, a Megolm session can be encrypted and authenticated using AES-256 in CBC mode. Figure 4.2 gives a schematic overview on how Alice computes keys to encrypt her Megolm session and to decrypt Bob’s Megolm session.

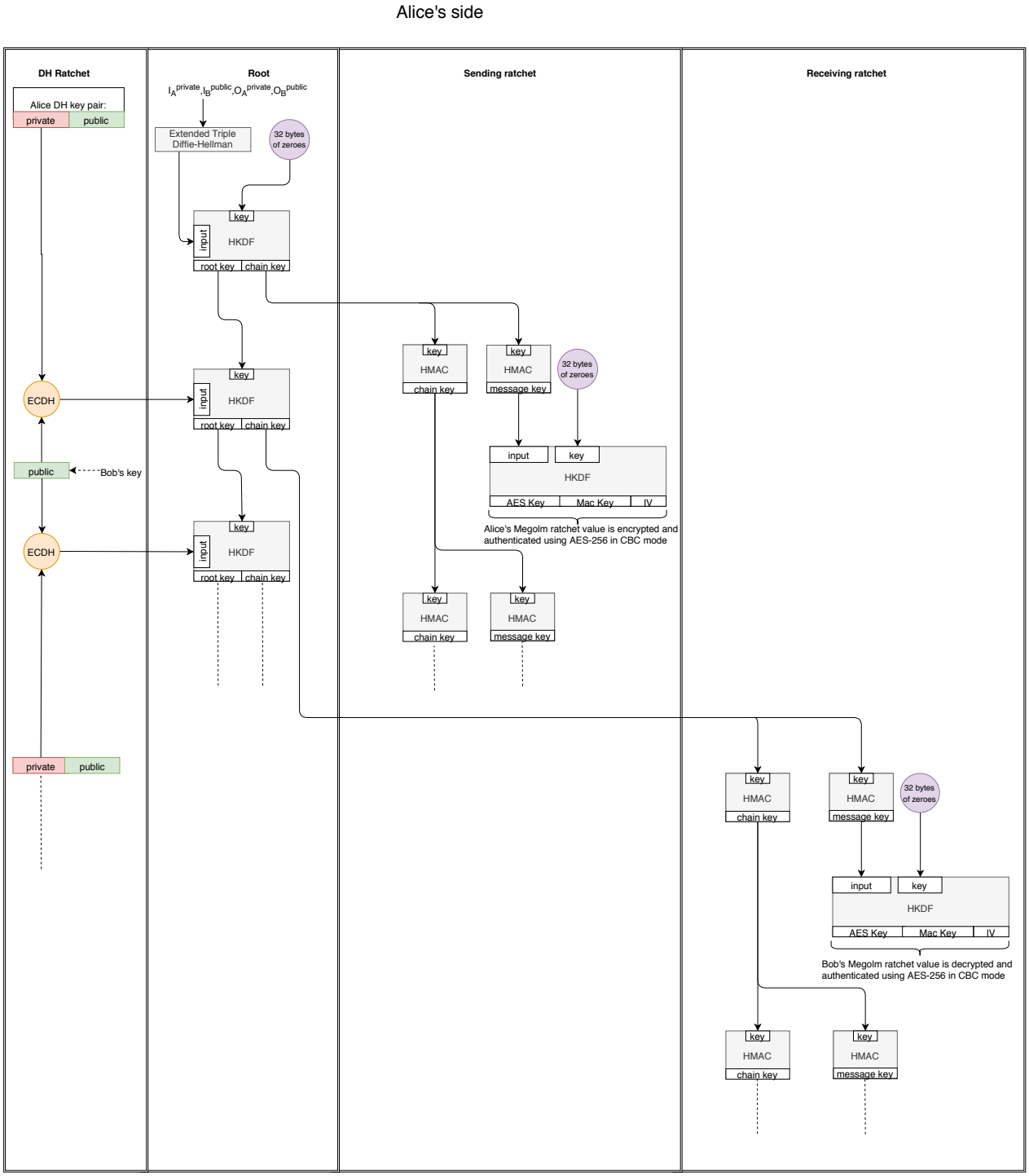

Figure 4.2: schematic diagram of the Olm protocol

4.2.2 Signing one-time keys

The one-time keys that are used in the Triple-Diffie Hellman (see Section 2.3.7) can be signed. This is not mentioned when defining the Olm algorithm because signing the one-time keys has both advantages and disadvantages. Assume Alice wants to establish an Olm session with Bob and have their device keys verified. Alice requests Bob's keys and Bob's homeserver sends his public identity key I_B^{public} and O_B^{public} to Alice. When the one-time key is unsigned, Alice does not know if the one-time key is actually from Bob from some adversary. A "weak forward secrecy attack" is then possible [34]. Meaning that an adversary can decrypt the first sent messages until a message is send back. Assume Eve is a man in the middle between Bob and the server (this is possible because the client-server API supports a HTTP connection). If Bob uploads the public keys, Eve can replace Bob's unsigned one-time key O_B^{public} with her own O_E^{public}. Alice will receive I_B^{public} and O_E^{public}. With these keys she computes:

$$S = ECDH\left(I_A^{private}, O_E^{public}\right) \parallel ECDH\left(O_A^{private}, I_B^{public}\right) \parallel ECDH\left(O_A^{private}, O_E^{public}\right).$$

If Eve wants to compute the same key S, she has to calculate:

$$S = ECDH\left(I_B^{private}, O_A^{public}\right) \parallel ECDH\left(O_E^{private}, I_A^{public}\right) \parallel ECDH\left(O_E^{private}, O_A^{public}\right).$$

If Eve compromises Bob's identity key she can decrypt Alice's Megolm sessions and consequently, decrypt Alice's messages meant for Bob until Bob decides to send a message to Alice. If Bob sends a message back he also sends the public part of his ratchet key T_i which is used to advance the root key. Eve can't access Bob's private ratchet key and is therefore unable to continue to compromise keys any further.

If Bob's one-time key O_B^{public} is signed with Bob's long term private fingerprint key then Eve is not able to replace O_B^{public}. This is due to the fact that Eve does not have Bob's private key to sign her one-time key since it is stored on Bob's client. However, the downside of signing Bob's public one-time key is the lack of deniability [24]. If Alice hands over the transcript between her and Bob and her identity key to a third party, she can prove that Bob genuinely is Bob in the transcript. Alice can prove this as follows:

If Bob wants to send an encrypted message to Alice, he first has to send an encrypted Megolm session to Alice. To encrypt the session, he has to be able to generate a key from the shared key S which contains his signature. Bob sends the encrypted session to his homeserver.

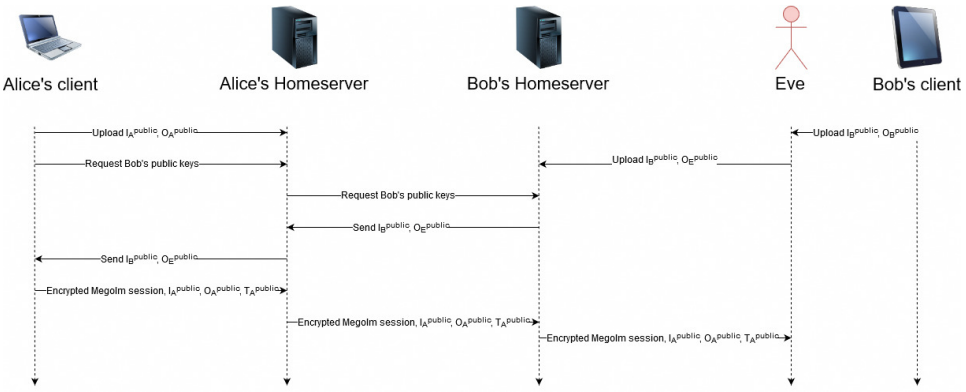

Figure 4.3: Potential scenario where bob's one time key is not signed

Bob's homeserver then signs the encrypted session with its Ed25519 key BHS. After appending the signature, Bob's homeserver sends it to Alice's homeserver. Alice can now prove that she received a signed encrypted Megolm session from Bob's homeserver. She can also prove that the encrypted Megolm session is constructed with a key from Bob's signed one-time key, as shown in Figure 4.4. Without the signed one-time key Bob could still deny the transcript because Bob could say that it was another client or device connected to his server that he did not know of.

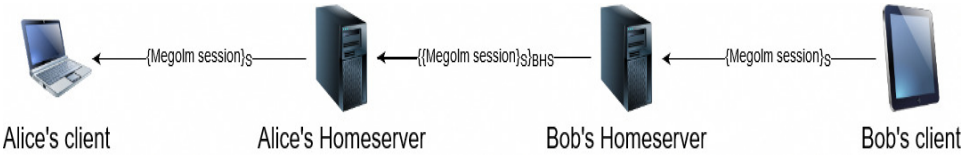

Figure 4.4: Schematic diagram of the encrypted and signed Megolm session

Since there are both advantages and disadvantages when signing the one-time key, Matrix has left this option to the client developers. Riot, the client developed by The Matrix.org Foundation C.I.C., has chosen to sign the one-time keys [44].

4.2.3 The Megolm algorithm

Creating a Megolm session starts by generating a random 1024 bit string R_i, where i is the number of times the ratchet value has iterated. This bit string can be divided into four 256 bit strings $R_{i,j}$, where $j \in \{0, 1, 2, 3\}$.

These values are advanced as follows:

$$R_{i,0} = \begin{cases} H_0\left(R_{2^{24}(n-1),0}\right) & \text{if } \exists n | i = 2^{24}n, \\ R_{i-1,0} & \text{otherwise,} \end{cases}$$

$$R_{i,1} = \begin{cases} H_1\left(R_{2^{24}(n-1),0}\right) & \text{if } \exists n | i = 2^{24}n, \\ H_1\left(R_{2^{16}(m-1),1}\right) & \text{if } \exists m | i = 2^{16}m, \\ R_{i-1,1} & \text{otherwise,} \end{cases}$$

$$R_{i,2} = \begin{cases} H_2\left(R_{2^{24}(n-1),0}\right) & \text{if } \exists n | i = 2^{24}n, \\ H_2\left(R_{2^{16}(m-1),1}\right) & \text{if } \exists m | i = 2^{16}m, \\ H_2\left(R_{2^{8}(p-1),2}\right) & \text{if } \exists p | i = 2^{8}p, \\ R_{i-1,2} & \text{otherwise,} \end{cases}$$

$$R_{i,3} = \begin{cases} H_3\left(R_{2^{24}(n-1),0}\right) & \text{if } \exists n | i = 2^{24}n, \\ H_3\left(R_{2^{16}(m-1),1}\right) & \text{if } \exists m | i = 2^{16}m, \\ H_3\left(R_{2^{8}(p-1),2}\right) & \text{if } \exists p | i = 2^{8}p, \\ H_3\left(R_{i-1,3}\right) & \text{otherwise.} \end{cases}$$

where:

$$H_0(A) = HMAC(A, \text{“0x00”}),$$
$$H_1(A) = HMAC(A, \text{“0x01”}),$$
$$H_2(A) = HMAC(A, \text{“0x02”}),$$
$$H_3(A) = HMAC(A, \text{“0x03”}).$$

As seen above, during the first $2^8 - 1$ iterations only the 256 bit string $R_{i,3}$ is advanced, the other parts remain at their initial value. Every 2^8 iterations, first $R_{i,3}$ is reseeded from $R_{i,2}$ and then $R_{i,2}$ is advanced. Every 2^{16} iterations, first $R_{i,3}$ and $R_{i,2}$ are reseeded from $R_{i,1}$ and then $R_{i,1}$ is advanced. Every 2^{24}, first $R_{i,3}$, $R_{i,2}$ and $R_{i,1}$ are reseeded from $R_{i,0}$ and then $R_{i,0}$ is advanced.

After advancing the ratchet values, they are concatenated back into a 1024 bit string which will be used as a message key M_i.

$$M_i = R_{i,0} \parallel R_{i,1} \parallel R_{i,2} \parallel R_{i,3}$$

Using HKDF with as input the 1024-bit message key and a zero-string of 32 bytes as salt, the cipher key, MAC key and IV can be derived.

When the developers initially designed Megolm they used reseedings so that a Megolm session could be used indefinitely. However, when the NCC Group conducted a security audit on Megolm, they warned for the lack of backward secrecy [7]. To help mitigate this problem, a Megolm session rotates every 100 iterations or after 1 week [45]. Consequently, the algorithm does not make use of the reseeds and only relies on the advancing 256 bit $R_{i,3}$ when generating new keys. See Figure 4.5 for a schematic overview of the Megolm ratchet.

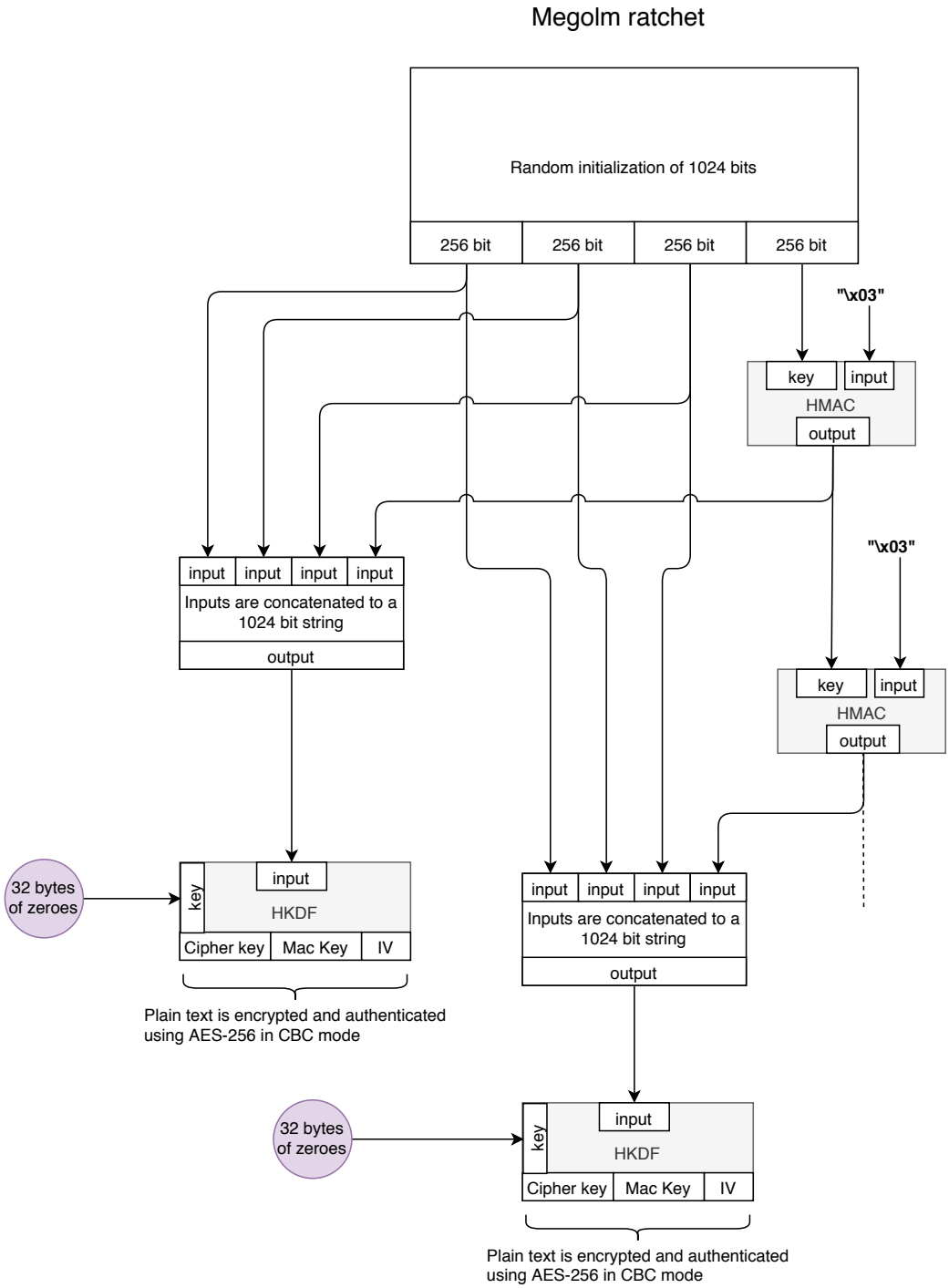

Figure 4.5: Schematic diagram of the Megolm protocol

4.2.4 The protocol for encrypting and authenticating a message

Assume Alice and Bob want to communicate with each other via Matrix. They first verify each other's public fingerprint key using SAS verification (see Section 4.1). If successful, they compute a key to end-to-end encrypt and authenticate a message as follows:

1. Alice's and Bob's clients first generate a one-time key pair using Curve25519 (see Section 2.3.5): O_A and O_B.
2. Alice and Bob publish the public part of their identity key pair I (signed with their fingerprint key) and one-time key generated keys to their homeserver.
3. Alice wants to connect to Bob so she requests Bob's public keys.
4. Bob sends Alice I_B^{public} and O_B^{public}.
5. Using $I_A^{private}$, $O_A^{private}$, I_B^{public}, O_B^{public} and triple Diffie-Hellman (see Section 2.3.7) she computes the shared secret S.
6. Alice then uses S, the context string and a salt to compute the root key R_0 and chain key $C_{0,0}$:

   $$R_0 \parallel C_{0,0} = HKDF(0, S, \text{“OLM\_ROOT”}, 64).$$

7. With $C_{0,0}$ she can compute the first message key:

   $$M_{0,0} = HMAC\left(C_{0,0}, \text{“\textbackslash x01”}\right).$$

8. The 256 bit AES key, 256 bit HMAC key, and 128 bit AES IV are derived from the message key:

   $$AES\_KEY_{0,0} \parallel HMAC\_KEY_{0,0} \parallel AES\_IV_{0,0} = HKDF\left(0, M_{0,0}, \text{“OLM\_KEYS”}, 80\right).$$

9. Alice then constructs her Megolm session which consists of three parts:
   - a 32 bit counter, i,
   - a newly computed Ed25519 key pair, K,
   - a ratchet R_i, which consists of four 256 bit values, $R_{i,j}$ where j $\in$ {0,1,2,3}.

Alice encrypts i, K and R_i with the AES key and AES IV from the Olm session. She also creates a ratchet key pair T_A so Bob and Alice can advance the root key later on. Then the encrypted text, chain index, T_A^{public}, I_A^{public} and O_A^{public} is sent to bob.

10. When Bob receives T_A^{public}, I_A^{public} and O_A^{public}, Bob is able to go through steps 5-9 and compute the same symmetric AES key to decrypt Alice's Megolm ratchet, counter and key pair.

11. Bob creates his own ratchet key pair T_B. Bob uses T_A^{public} and $T_B^{private}$ to construct a new shared secret using Curve25519. To advance the root key and to create a new chain key the HKDF is used with as input the previous root key and the new constructed shared secret:

$$R_1 \parallel C_{1,0} = HKDF\left(R_0, ECDH\left(T_B^{private}, T_A^{public}\right), \text{“OLM\_RATCHET”}, 64\right).$$

12. Using the newly computed chain key, $C_{1,0}$, Bob computes a new chain and message key in the same way Alice did.

13. With the new message key Bob derives the 256 AES key, 256 bit HMAC key, and 128 bit AES IV.

14. Bob now initializes randomly 1024 bits to create a Megolm session and encrypts this session with the AES key and IV, authenticated by the 256 bit HMAC key.

15. Bob sends his encrypted text, chain index and T_B^{public} to Alice so she can decrypt it.

16. Since they now have their own and each other Megolm session they can encrypt and authenticate their messages and send it to each other via an unsecure channel.

4.2.5 Complexity of encrypted group chats

Sessions are created lazily. That means that only when a client wants to send a message it creates its own Megolm session. If N devices are talking in an encrypted room with N devices then there are $N(N-1)/2$ Olm sessions because every device in the room has to create an Olm session with any other device in the room. There are also at least N Megolm sessions, since every device has its own Megolm session to encrypt their messages with. If only M devices write messages out of the N devices in an encrypted room then there are $N \cdot M + M(M-1)/2$ Olm

sessions and at least M Megolm sessions.

Olm sessions can also be reused. If Alice and Bob are talking to each other in multiple rooms the same Olm session can be used.

4.3 Key sharing and key backup

Say Alice is talking to Bob via her desktop client. She logs into the Riot app on her mobile phone for the first time. Alice is not able to read the conversation on her mobile between her and Bob since she uses a new and unverified device. She may want to have access to the messages she has sent via her desktop client. If so, she can request keys from her desktop client by sending a m.room_key_request to all her other devices in the room. If the desktop client receives that message and it wants to share the keys with her mobile phone, the keys can be exported via Olm.

When a Matrix user logs out of his/her client, all the Megolm sessions and device keys are discarded. This is due to the fact that if an adversary steals the device, the adversary is not able to retrieve the keys unencrypted from the storage of the device. Therefore, if a Matrix user wants to backup his/her Megolm sessions before logging out, the user can export the Megolm sessions from the client to a file. The file is encrypted using a new password that the user supplies:

1. The Megolm sessions are encoded as a JSON object, see Figure 4.7 for more information.
2. The user generates a password to compute two 256 bit keys using $K \parallel K' = PBKDF2\left(HMAC\text{-}SHA\text{-}512, \text{password}, S, N, 512\right)$

   where:

   - N is the number of rounds, where $N \geq 100000$,
   - S is a random 128 bit salt,
   - 512 is the length of the output key,
   - K is the AES-256 key,
   - K' is the HMAC-SHA-256 key.

   See Figure 4.6 for more information.
3. Serialize the JSON object as a UTF-8 string and encrypt it using the key that AES-CTR-256(IV, K) produces. Here, IV is a random 128 bit string.

4. Concatenate the encrypted JSON object C, S, IV, N and the value of $HMAC\text{-}SHA\text{-}256(K', C, IV, N)$.

5. Encode the above string with Base64.

6. Prepend the resulting string with "—BEGIN MEGOLM SESSION DATA—" and append the "—END MEGOLM SESSION DATA—". The file is now ready to exported and can be imported by other devices that use Matrix.

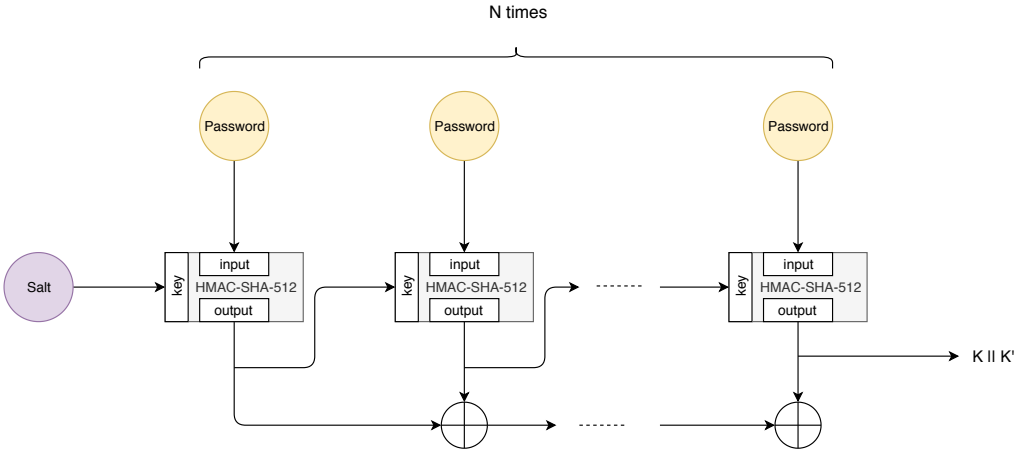

Figure 4.6: Schematic diagram of pbkdf2, based on [36]

Key export format

The exported sessions are formatted as a JSON array of `SessionData` objects described as follows:

`SessionData`

Parameter	Type	Description
algorithm	string	Required. The encryption algorithm that the session uses. Must be `m.megolm.v1.aes-sha2`.
forwarding_curve25519_key_chain	[string]	Required. Chain of Curve25519 keys through which this session was forwarded, via `m.forwarded_room_key`_ events.
room_id	string	Required. The room where the session is used.
sender_key	string	Required. The Curve25519 key of the device which initiated the session originally.
sender_claimed_keys	{string: string}	Required. The Ed25519 key of the device which initiated the session originally.
session_id	string	Required. The ID of the session.
session_key	string	Required. The key for the session.

Figure 4.7: Key export format, taken from [2]

Chapter 5

Related Work

The topic of key management in decentralised chat services has been researched before. However, how the keys are managed and with what purpose differs immensely between each decentralised chat service. One of the researched decentralised chat services is ChronoChat [46]. In contrast to Matrix, ChronoChat is based on Named Data Networking (NDN). NDN makes it possible to route data based on name instead of IP address by binding the name and content of every packet through a digital signature [47]. The network architecture of NDN enables to have peer to peer communication and can be run serverless [46]. However, the downside is that the key management of ChronoChat only focuses on public key authentication and lacks end-to-end encryption, it is therefore not comparable with Matrix.

Matrix has also been compared to other centralized end-to-end encrypted chat services in terms of security and user experience [37], [28]. The papers define cryptographic properties and verify whether the chat services meet them. They also propose how Matrix can improve security by introducing two-step verification and screen security (not allowing screenshots of the chat messages).

Chapter 6

Discussion

When examining the key management of Matrix, it became evident how Matrix realises a secure and scalable end-to-end encryption in a decentralized network. However, there still exist areas for further enhancement. Some of these areas are currently in development by The Matrix.org Foundation C.I.C.

6.1 Improvements to Matrix that are currently in development

6.1.1 Traffic data vulnerabilities

The key management in Matrix makes it possible to securely end-to-end encrypt the contents of a message. The encryption guarantees confidentiality, integrity and forward secrecy [7]. However, this does not directly mean that the communication flow between Matrix users is secure. As seen in Figure 3.3, a lot of unencrypted traffic data is sent along with the encrypted message. Sirius, a project created by Europol, researched what the European Intelligence services found the most important type of data [4]. The report concluded that traffic data is significantly more interesting to research for European intelligence services, than the actual content.

As seen in the message and state events, traffic data disclose a lot about the Matrix users. Server timestamps, room identification, the time of how long the message has been in transit (age), device identification, the sender of the event is all unencrypted. As a result of the architecture of Matrix, this unencrypted traffic data is necessary and cannot simply be omitted, because homeservers need to verify these events and

keep track of the state of the room.

Due to the unencrypted traffic data, Matrix is very susceptible for traffic data analysis. To solve this, Matrix is working on a peer-to-peer network architecture [22] which includes "The Loopix Anonymity System" [41]. It provides cover traffic (dummy traffic) and poisson mixing (delaying messages randomly).

6.1.2 Cross signing

Another disadvantage of the key management in Matrix is the number of times devices have to be verified. For N devices in a room there needs to be $N(N-1)/2$ device verifications to verify the whole room. That means to verify all devices in a room with 10 Matrix users that have 2 devices each, would require $20(20-1)/2 = 190$ verifications. This is time consuming, and many users criticize Matrix for this [23]. To solve this issue, the Matrix developer team has been working on implementing cross-signing since December 2018 [12]. It is based on a proposal made by one of the developers [11].

The idea is that each user has three key pairs:

- A “self-signing” Ed25519 key pair to sign their own devices.
- A “user-signing” Ed25519 key pair to sign master keys of other users.
- A “master” Ed25519 key pair for identification and signing their self-signing and user-signing key pairs.

The public part of the keys is then published to the homeserver so that the other party is able to retrieve them. Assume Alice and Bob want to verify their devices with each other. Just like the current device verification, they meet with each other via an out-of-band channel such as physically or a VoIP call. With SAS (see Section 4.1.1) they verify each other’s master keys. Alice’s and Bob’s devices can then trust each other when:

- Alice and Bob’s master keys have signed their own user-signing and self-signing keys.
- Alice and Bob’s self-signing keys sign their own devices.
- Alice and Bob sign with their user-signing key the other party’s master key (see Figure 6.1).

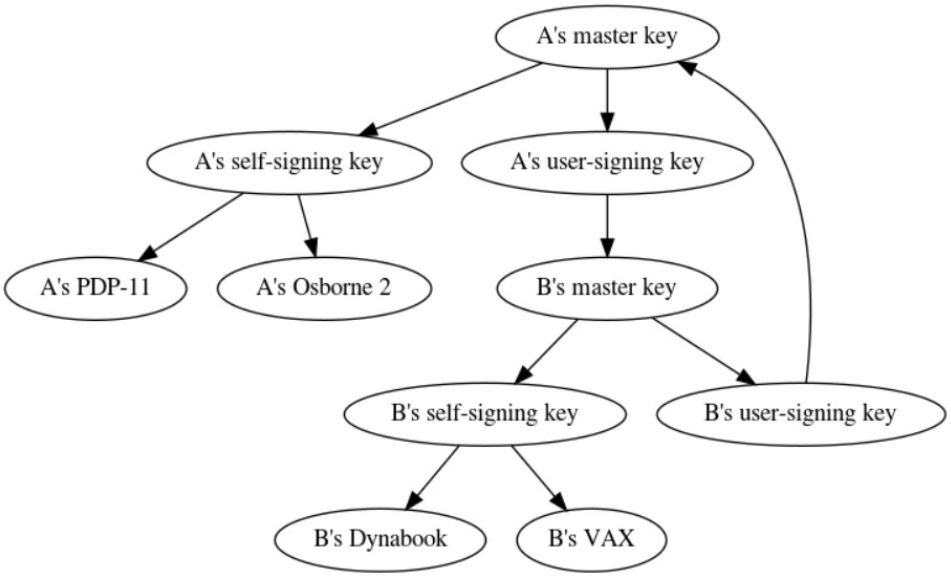

Figure 6.1: Schematic example of the cross signing proposal, taken from [11]

6.2 Recommendations for improving key management in Matrix

There were 2 areas identified of the key management in Matrix, where The Matrix.org Foundation C.I.C. is not yet working on, nor has a proposal to improve it. These are recommendations to make the key management of Matrix more secure:

6.2.1 Lack of Deniability

One way to guarantee key deniability would be omitting the one-time key signatures. However, the downside of this is that it would enable weak forward secrecy. Another option would be implementing “Deniable authenticated key exchange with Zero-knowledge” (DAKEZ), instead of Triple-Diffie-Hellman to establish shared secret keys. According to Unger and Goldberg [39], DAKEZ has both deniability and perfect forward secrecy.

6.2.2 Megolm ratchet

The Megolm ratchet randomly initializes a 1024 bit string. However, only 256 bits of the 1024 bit string are used to advance keys. Since the other 768 bits remain constant during the whole session, the input of the HKDF is mainly constant as well (see Figure 4.5). As a consequence, the security of forward secrecy is reduced. An improvement would be to advance the whole 1024 bits ratchet value instead of just 256 bits.

For example, create a Megolm session by generating a random 1024 bit string R_i, where i is the number of times the ratchet value has iterated. This bit string can be split into two bit strings of 512 bits: $R_{i,0}$ and $R_{i,1}$. Then by using HMAC with the hash function SHA-512 and some arbitrary input string, advance the values of $R_{i,0}$ and $R_{i,1}$ after each generated key:

$$R_{i,0} = HMAC(R_{i-1,0}, \text{“0x00”}),$$

$$R_{i,1} = HMAC(R_{i-1,1}, \text{“0x01”}).$$

Then similar to the regular Megolm algorithm, concatenate $R_{i,0}$ and $R_{i,1}$ back into a 1024 bit string which will be used as message key M_i:

$$M_i = R_{i,0} \parallel R_{i,1}.$$

Using HKDF with the message key as input, the Cipher key, MAC key and IV can be derived.

Now, if a cipher key is compromised, calculating the previous established keys is computationally much more difficult, since the value of the whole 1024 bit string is advanced, instead of just 256 bits.

Chapter 7

Conclusions

This thesis conducted an analysis about the key management of Matrix. It examined how keys in Matrix are computed, verified, used, exchanged and stored and how these areas relate to one another. Key generation starts when a Matrix user logs into their client for the first time on a client supported platform. Matrix registers this as a “device”. Each device has two long-term key pairs generated, an Ed25519 fingerprint key pair and a Curve25519 identity key pair. If two Matrix users want to authenticate each other before communicating, they can verify their device’s fingerprint key pair by computing a SAS. By using a fingerprint signature on the identity key, the public identity key can be verified.

After verifying the long-term device keys, Matrix users are able to compute a key that is used to end-to-end encrypt a message. To compute such a key, first an Olm session needs to be established to create a secure channel between two Matrix users. An Olm session is based on the double ratchet algorithm of Signal. The Olm algorithm consists of two ratchets, a symmetric key ratchet to compute keys to encrypt and decrypt Megolm sessions, and a Diffie-Hellman ratchet that advances each time a Megolm session has been exchanged between two Matrix users’ clients. If the Diffie-Hellman ratchet advances, it replaces the symmetric key ratchet with a new symmetric key ratchet (see Figure 4.2). Next, both Matrix users’ clients compute a Megolm session, which is a randomly initialized single ratchet. They exchange their Megolm session with each other using the Olm keys. If a message is going to be send, the ratchet advances and a new key is computed to symmetrically encrypt the message. Since the receiver has the same ratchet and the encryption method is symmetric, the receiver can compute the same key and decrypt the message.

Keys can also be shared among other devices. If a Matrix user is unable to decrypt a conversation (because for example the user logs into a client for the first time), the client can request keys from the other user's devices, and the keys can be exported via Olm.

If a Matrix user logs out of their client, the keys are discarded due to security reasons (see Section 4.3). To store the keys, the Matrix user has the option to export the Megolm ratchets to a file, which is secured by a user generated password.

Matrix's end-to-end encryption generally is a secure algorithm that complies with most security properties [37]. For example, it guarantees authentication since Matrix uses SAS verification (see Section 4.1.1), which is considered to be a secure way to verify the fingerprint key of a device and to counter an unknown key-share attack. However, it requires an extensive amount of time to verify each fingerprint key in a room with many participants. A Matrix developer is therefore currently working on cross signing, so that a Matrix user only has to sign their own devices, and the other party's master key. This reduces the amount of SAS key verifications in a room.

One of the security properties that Matrix does lack, is key deniability. The developers of Matrix state that the Olm protocol with signed one-time keys still has partial deniability [24]. However, that statement is invalid since they did not keep in mind that homeservers also sign the events (see Section 4.2.2). Due to the signatures on the one-time keys by the client and the signature of the homeserver, a message transcript in the hands of a third party cannot be denied by the transcript participant.

Furthermore, while end-to-end encryption of Matrix secures the content of the message sufficiently, it does not directly mean that the message itself is secure. Matrix leaks a significant amount of unencrypted metadata which still can disclose information about the message and the sender.

7.1 Future work

An interesting research avenue would be to explore the implementation of DAKEZ to potentially improve the deniability of Matrix's key management. Another research opportunity would be to analyse how much the traffic data discloses about Matrix users, and find an intermediate solution until The Matrix.org Foundation C.I.C. releases peer-to-peer Matrix with The Loopix Anonymity System.

Bibliography

[1] Clients matrix. https://matrix.org/clients/.

[2] End-to-end encryption implementation guide. https://github.com/matrix-org/matrix-doc/blob/master/specification/modules/end_to_end_encryption.rst#key-export-format.

[3] Matrix specification. https://matrix.org/docs/spec/.

[4] Sirius: European union digital evidence situation report 2019, December 2019. https://www.europol.europa.eu/newsroom/news/sirius-european-union-digital-evidence-situation-report-2019.

[5] Latest Updates 2014-01-28 and Base64, January 2014. http://articles.squaredprogramming.com/2014/01/i-had-been-spending-lot-of-time.html.

[6] Micah Alcorn. Introducing origin messaging: Decentralized, secure, and auditable, August 2018. https://medium.com/originprotocol/introducing-origin-messaging-decentralized-secure-and-auditable-13c16fe0f13e.

[7] Alex Balducci and Jake Meredith. Olm cryptographic review, 2016. https://www.nccgroup.trust/us/our-research/matrix-olm-cryptographic-review/.

[8] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures, September 2011. https://ed25519.cr.yp.to/ed25519-20110926.pdf.

[9] Daniel J. Berstein. Curve25519: new Diffie-Hellman speed records, 2006. https://cr.yp.to/ecdh/curve25519-20060209.pdf.

[10] Bill Buchanan. Elliptic curve diffie hellman (ecdh) with differing elliptic curves. https://asecuritysite.com/encryption/ecdh3.

[11] Hubert Chathi. Cross-signing devices with device signing keys, December 2018. https://github.com/uhoreg/matrix-doc/blob/cross-signing2/proposals/1756-cross-signing.md.

[12] Hubert Chathi. Msc1756: cross-signing devices using a master identity key, December 2018. https://github.com/matrix-org/matrix-doc/pull/1756.

[13] The Matrix.org Foundation C.I.C. An open network for secure, decentralized communication, 2019. https://matrix.org/.

[14] Katriel Cohn-Gordon, Cas J. F. Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees, 2017. https://www.semanticscholar.org/paper/On-Ends-to-Ends-Encryption%3A-Asynchronous-Group-with-Cohn-Gordon-Cremers/4fbf0992604fa0fde4ab55ed09f7b52b44c30b83.

[15] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael, March 2003. https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf#page=1.

[16] Johan Dams. An introduction to elliptic curve cryptography, October 2012. https://www.embedded.com/an-introduction-to-elliptic-curve-cryptography/.

[17] Quynh Dang. Recommendation for applications using approved hash algorithms, August 2012. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-107.pdf.

[18] William F. Ehrsam, Carl H. W. Meyer, John L. Smith, and Walter L. Tuchman. Message verification and transmission error detection by block chaining, 1976.

[19] Josh Fraser. An Update Regarding Origin Messaging, March 2019. https://medium.com/originprotocol/an-update-regarding-origin-messaging-1aab21f586c8.

[20] Darrel Hankerson, Scott Vanstone, and Alfred Menezes. The state of elliptic curve cryptography, 2004. http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=2A080B94753574E1EFCDECBC94630457?doi=10.1.1.206.1573&rep=rep1&type=pdf.

[21] T. Hansen. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF), May 2011. https://tools.ietf.org/html/rfc6234.

[22] Matthew Hodgsen. The 2019 matrix holiday update!, December 2019. https://matrix.org/blog/2019/12/24/the-2019-matrix-holiday-update/.

[23] Matthew Hodgson. Support transitive trust for e2e verification between users (i.e. web-of-trust) in some situations, February 2019. https://github.com/matrix-org/matrix-doc/issues/1886.

[24] Richard van der Hoff. Signature keys and user identity in libolm, 2016. https://gitlab.matrix.org/matrix-org/olm/blob/master/docs/signing.md.

[25] Richard van der Hoff, Mark Haines, and Matthew Hodgson. Megolm group ratchet, 2016. https://gitlab.matrix.org/matrix-org/olm/blob/master/docs/megolm.md.

[26] R. Housley. Cryptographic message syntax (cms), September 2009. https://tools.ietf.org/html/rfc5652#section-6.3.

[27] ECMA International. The json data interchange syntax, December 2017. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[28] Christian Johansen, Aulon Mujaj, Hamed Arshad, and Josef Noll. Comparing implementations of secure messaging protocols (long version), November 2017. https://www.duo.uio.no/handle/10852/60949.

[29] S. Josefsson. Edwards-curve digital signature algorithm (eddsa), January 2017. https://tools.ietf.org/html/rfc8032#section-3.

[30] Daniel Kline. Elliptic curves and cryptography, August 2016. http://math.uchicago.edu/~may/REU2016/REUPapers/Kline.pdf.

[31] H. Krawczyk. HMAC-based Extract-and-Expand Key Derivation Function (HKDF), May 2010. https://tools.ietf.org/html/rfc5869.

[32] H. Krawczyk, M. Bellare, and R. Canetti. Hmac: Keyed-hashing for message authentication, 1997. https://tools.ietf.org/html/rfc2104.

[33] Richard van der Hoff Mark Haines and Matthew Hodgson. Olm: A cryptographic ratchet. https://gitlab.matrix.org/matrix-org/olm/blob/master/docs/olm.md.

[34] Moxie Marlinspike and Trevor Perrin (editor). The x3dh key agreement protocol, 2016-11-04. https://signal.org/docs/specifications/x3dh/.

[35] Moxie Marlinspike and Trevor Perrin (editor). The double ratchet algorithm, 2016-11-20. https://signal.org/docs/specifications/doubleratchet/.

[36] K. Moriarty. Pkcs #5: Password-based cryptography specification, January 2017. https://tools.ietf.org/html/rfc8018#section-5.2.

[37] Aulon Mujaj. A comparison of secure messaging protocols and implementations, Spring 2017. https://www.mn.uio.no/ifi/english/research/groups/psy/completedmasters/2017/Aulon_Mujaj/aulon_mujaj_msc_comparison_secure_messaging_protocols_implementations_2017.pdf.

[38] Randall Munroe. Chat Systems, 2017. https://xkcd.com/1810/.

[39] Nik Unger and Ian Goldberg. Improved strongly deniable authenticated key exchanges for secure messaging, 2018. https://www.petsymposium.org/2018/files/papers/issue1/paper12-2018-1-source.pdf.

[40] J. Oikarinen and D. Reed. Internet relay chat protocol, May 1993. https://tools.ietf.org/html/rfc1459.

[41] Ania M. Piotrowska and Jamie Hayes. The loopix anonymity system, 2017. https://www.usenix.org/node/203839.

[42] Richard van der Hoff and Matthew Hodgsen, 2019. https://matrix.org/docs/spec/server_server/r0.1.3.

[43] S. Loreto and P. Saint-Andre and S. Salsano and G. Wilkins). Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP, April 2011. https://tools.ietf.org/html/rfc6202.

[44] Richard van der Hoff. Sign one-time keys, and verify their signatures, 2016. https://github.com/matrix-org/matrix-js-sdk/pull/243.

[45] Richard van der Hoff and Matthew Hodgsen. Client-server api, 2019. https://matrix.org/docs/spec/client_server/latest#m-room-encryption.

[46] Yingdi Yu, Zhenkai Zhu, Alexander Afanasyev, and Lixia Zhang. An endorsement-based key management system fordecentralized ndn chat application, 2014. http://named-data.net/wp-content/uploads/2014/07/ndn-tr-23-chronochat-security.pdf.

[47] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, Kc Claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. Named data networking, July 2014. https://dl.acm.org/doi/pdf/10.1145/2656877.2656887?download=true.