Teemu Makkonen

Implementing Encryption for Qt-based Matrix Client

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor’s Thesis

30 November 2019

Author Title Number of Pages Date	Teemu Makkonen Implementing Encryption for Qt-based Matrix Client 52 pages 30 November 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Juha-Pekka Kämäri, Lecturer

The objective of this thesis was to design and implement encryption for a messaging application that uses Matrix Open Standard to deliver the messages. The study was purposed to be used as a solid base that can be expanded upon by creating the implementation in such a way that it is easy to change if another encryption scheme would be used.

For the purpose of the thesis it was necessary to look in to the history of encryption and modern cryptography in order to gain more knowledge and to choose and design the proper encryption scheme that would fill out the necessary criteria set by the company. The bit by bit operations of the mostly used encryption schemes were studied as well as modern larger scale algorithms that provide more security around the strong encryption.

Matrix Open Standard needed to be studied as well for the understanding of how the messages are delivered. The Qt cross-platform framework was used for the development and the messaging application was running on a QEMU virtual emulator, running the company's own secure mobile operating system.

The implementation was successful and the goals of encrypting and decrypting messages sent with the application was achieved. The company was happy with the results although some parts were left without much attention. Overall the design and implementation gives good base to build upon or use it as is for future developments.

The cryptography knowledge gathered during this thesis gave the company a lot of confidence as to the cryptographic schemes explored in this study. It also gave them more understanding on their future evaluations on different encryption schemes implemented in the present study.

Keywords	Cryptography, encryption, messaging, communications, security

Contents

List of Abbreviations

Qt	Cross-platform framework maintained by the Finnish Qt Company. Created for building applications for any architecture or operating system.
Key	In cryptography means a shared secret that contains information on how to decrypt ciphertext into readable format.
Plaintext	Message in readable form.
Ciphertext	Message in encrypted form.
OTP	One-time-pad. An old encryption scheme that provides unbreakable encryption. It is limited by its key size.
Public-key	The public part of a key pair that is used to encrypt messages for the person holding the other part of the key pair. Can be publicly shared without compromising the encryption.
Private-key	The secret part of a key pair. Used to decrypt and sign messages. Has the be kept as a secret and not shared with anyone.
Perfect secrecy	Term used in modern cryptography to indicate the encryption cannot be broken even if given unlimited resources to the adversary.
Forward secrecy	Is a feature in modern cryptography key agreement protocols that gives assurances that future messages will not be decipherable even if one session key is compromised.
Computational security	Is a definition used in modern cryptography to describe that an encryption cannot be broken in a reasonable timeframe given the current state of computation.

Symmetric-key	A key that is needed for both encryption and decryption. Meaning the recipient of a message needs the same key as the sender in order to communicate.
Stream cipher	An encryption method in symmetric-key cryptography, where the encryption is done to one bit at a time using a random stream of bits.
Block cipher	An encryption method where the plaintext is divided in to block and the blocks are then encrypted separately from the other blocks.
XOR	Exclusive or. Logical operation in cryptography and mathematics in general, that takes two inputs, 0 or 1 for example, and outputs 1 if they differ and 0 if they are the same.
IV	Initialization vector. Used as input data to alter the outcome of some algorithm. Shared between two communicating parties to arrive at the same key after the IV is inputted in to an algorithm.
ECB	Electronic code book. An operating mode for block ciphers, where each block is encrypted individually.
CBC	Cipher block chaining. An operating mode for block ciphers where the encrypted result of each block is fed as an IV in to the next block, creating a chain.
DES	Data Encryption Standard. A block cipher that was the first official standard for encryption in the US.
DES2	Double-DES. Next iteration of DES to make it more secure, where DES is used twice.
DES3	Triple-DES. The final iteration of DES, where the DES is used three times.

AES	Advanced Encryption Standard. After DES was deemed insecure AES was developed to be the new standard for encryption. It is a block cipher as well.
RSA	Rivest-Shamir-Adleman. Widely used public-key cryptosystem.
Diffie-Helmann Key Exchange	Is a mathematical way of exchanging cryptographic keys over a public network.
MAC	Message Authentication Code. A short piece of data usually in the header of the actual data used to authenticate the data.
VoIP	Voice-over-IP. Term used to describe calls that go over the internet network.
SDK	Software Development Kit. A collection of software used to develop and build other software such as applications.
HTTP	HyperText Transfer Protocol. The underlying protocol of the world wide web.
API	Application Programming Interface. Interface used by servers to serve clients with data or to collect data that clients send to servers.
JSON	JavaScript Object Notation. A way of handling and storing data in a human readable form. Used widely in webservices.
REST	Represantional State Transfer. Architecture many servers on the internet use for storing data. Operates on top of HTTP.
KDF	Key Derivation Function. A term used in cryptography for a function that takes some input and generates a key.
QEMU	Short for Quick EMUlator. It is a lightweight open source virtual emulator for emulating different environments and operating systems.

Elliptic Curve	A mathematic function that defines a curve over a plane. Used often in everything related to cryptography, authentication, key generation and key agremeents.
NaCl	An algorithm designed to be used for encrypted messaging applications. Often called libsodium.
GUI	Graphical User Interface. Term used to describe the part of an application that is visible to the user.
CIA	Central Intelligence Agency. One of the main intelligence agencies in the US. Tasked with mostly foreign intelligence.
UML	Unified Modeling Language is a modeling language used in software engineering to show the design of a large system.

1 Introduction

On the internet everything one does is sent over a network where hackers, companies and states take note and save everything. Their goals are different: they will try to steal credit card information, extract every piece of data point in order to serve better ads or they try and identify political standings and thoughts in the name of national security. What ever their end goal is, its invading privacy.

Today encryption is added to everything from browsers and web traffic to local hard drives and communications. But what is encryption? How does it work? Why is it good for privacy? How can anyone be sure their personal messages aren’t analyzed by large tech companies or evil adversaries? How is encryption actually created? These are the questions this thesis tries to answer.

This thesis starts with the history of encryption and take a deep dive into modern cryptography to understand the underlying mathematics of encryption. It does shine a light into the bit by bit operations encryption does to data and create understanding of how mathematics can create security. Then it takes a look at modern algorithms that secure todays messaging applications and delve into the architecture of one upcoming, open source and decentralized messaging protocol called Matrix Open Standard.

Finally the goal is to showcase a concrete implementation of modern encryption on a messaging application that uses Matrix for transportation of messages. In the process the thesis outlines what needs to be considered on the broader spectrum when designing encryption and what are the intricate details when implementing it. The application itself has been created with Qt cross-platform framework which requires understanding of the C++ coding language and the Qt framework in order to develop the implementation.

The study was done for a Finnish cybersecurity company, Necuno Solutions Ltd. The company sells secure communications solutions for industries that require high level of security such as investigative journalism, lawfirms, healthcare, critical infrastructure, governmental agencies and defense. The groundwork laid by this engineering thesis is intended to be used for their future products.

2 Encryption

Encryption in its most basic definition is taking some data and scrambling it in a way that only the authorized parties are able to understand the information. This, in most common and historic encryption schemes required both parties to know a specific key that can be used to encrypt and decrypt the information. A more concrete explanation would be taking a some message often referred to as plaintext like “Hello world”, making it unreadable, referred to as ciphertext, like “SGVsbG8gd29ybGQK” and giving it to another person who knows the secret key and can use it to decrypt it back to its original meaning.

Encryption has been prevalent through history and it starts all the way from 500 BC with simple substition ciphers [1, p. 2], to mathematically complex modern encryption schemes and even modern quantum resistant encryption. This chapter does go over the brief history of encryption..

People have throughout history tried to keep some sensitive information hidden from unintendent persons often using some form of encryption. One of the earliest forms of encryption was substitution ciphers, where a word is taken and the letters are substituted by shifting the alphabet by a predefined amount. This required the recipient of the encrypted information to know the number or “key” for example to shift by 3 to decrypt the message. The most well known cipher of this kind is called Caesar’s Cipher and was made famous by Julius Caesar, who used it to communicated with his generals. However Caesar used always a fixed key of 3 which was not very secure once the key was known [1, p. 2]. These substitution ciphers quickly became obsolete as they can be easily cracked using frequency analysis or a brute force attack. Brute force attack or exhaustive search is a way of trying every single possible key for the decryption until the decrypted plaintext makes sense. In the substitution cipher the key can only be 0-28 when using the Finnish alphabet, because there are 29 letters, resulting in 29 different variations and in turn is very vulnerable to brute force attack. Frequency analysis is a form of cryptanalysis and it is used to aid in decrypting of the ciphertexts by analyzing the frequency of certain letters and letter pairs. In every language some letters are more frequent than others and this can reduce the bruteforcing significantly.

During medieval times cryptography and encryption methods were not improved greatly. Most notably “polyalphabetic” ciphers were invented to fight the frequency analysis developed to crack simple substitution ciphers. In the 1917 the one-time-pad (OTP) was invented by Gilbert Vernam and Joseph Mauborgne and it is the only truly unbreakable cipher [2]. OTP works by having a encryption key the same size or larger than the actual message or plaintext consisting of truly random numbers. Then each letter is added to each number of the encryption key by using modular addition. The resulting ciphertext then has no relation to the original plaintext if the OTP is unknown. The recipient has to have the exact same key that was used to encrypt the original text in order to decrypt the ciphertext. The encryption key is to be destroyed after usage and never used again, hence the name “one-time-pad”.

Right before World War II the Germans invented the Enigma machine, it was an electromechanical encryption machine that used a rotor mechanism that scrambles the 26 letters of alphabet. It was extensively used to encrypt messages used by the Nazis in the World War II and great effort was used to decrypt those messages by the opposing forces. Enigma machine went through many iterations but the basic concept was that when a key is pressed a light above the keyboard would light up indicating a letter of the alphabet. One person would then type a message on the machine and another would write down the illuminated letters which would give out the ciphertext. The rotor inside the machine would rotate on every keypress changing the electrical connections between the keys and the lights. Entering the ciphertext on the receiving end would give out the original message. The security of the system was based on predefined settings which would change daily and distributed to each station beforehand. Enigma machines encryption was not perfect and could be decrypted with intensive mathematical cryptanalysis. Later on in the war The United Kingdom started the Ultra program to decrypt these messages and the infamous Turing machines was built during that time. The decryption of the Nazi information played a crucial part in the end of the war.

2 Modern Cryptography

Modern cryptography started after the World War II and it is defined by using computers to do encryption instead of mechanical machines. It is based on mathematical functions that the computers can calculate. Before modern cryptography, classical cryptography could be thought of as an art form where construction of good encryption was based on creativity and personal skill instead of rigorous mathematical theory. In the computer era however a rich theory and science emerged around cryptography allowing it to become more than just encryption of communications and secret messages. It opened up the possibilities for studying authentication, digital signatures, protocols for exchanging secret keys, electronic auctions and elections as well as digital currency. With modern cryptography it is often more appropriate to talk about the individual bits instead of the letters of the alphabet, like in the previous chapter of historic cryptography. So, in the following chapters, plaintext and ciphertext refer to 1's and 0's that the computer uses instead of the letters of the alphabet. The following chapters go over the two main approaches for modern cryptography, the Symmetric-key cryptography and the Asymmetric-key cryptography and some examples for these schemes are looked at in more detail.

2.2 Symmetric-key Cryptography

Symmetric-key cryptography is also known as private key cryptography, as it uses a predefined key that both parties have to know. Most historical ciphers could be thought of as being private key cryptography as they often used a key, known by both parties, for example the substitution cipher referred in Chapter 2. It used a key, a number, that was used to determine how much was the alphabet shifted for the encryption and decryption of the plaintext. Symmetric-key cryptography brings up some key concepts of cryptography in general. Those concepts are perfectly secret, often called perfect secrecy, and computational security.

In modern cryptography a perfect secrecy is achieved when given unlimited computing power the ciphertext cannot be decrypted by an adversary, due to the fact that the adversary trying to decrypt the ciphertext lacks enough information to succeed, regardless of the adversary's computational power. Such schemes are called perfectly secret or information-theoretically secret. [3, 29-35.]

Perfectly secret schemes are different from computationally secure schemes, which is the aim for most of the modern cryptography. Most modern encryption schemes can be theoretically broken given enough time or computational power, because they are not perfectly secret. However, the time and effort of breaking these schemes is astronomical and would require life times or even larger than universes age to succeed. The reason for most currently used encryption schemes to not be perfectly secret, is the practicality of it. The mathematical proof for perfect secrecy goes beyond the scope of this thesis. In the interest of this thesis it is sufficient to say that perfect secrecy can be mathematically achieved, but it lacks practicality as it is proven to only be achievable if the key length is the same or larger than the plaintext length and the key is then never used again. The one-time-pad is one such encryption scheme that is perfectly secret. [3, p. 58]

Computational security is often defined as an adversary having a very small probability to decrypt the message given a certain timeframe. That probability should be thought of to be small enough that it can safely be assumed it will never really happen. Modern encryption schemes are computationally secure to find a compromise between practicality and perfect secrecy. [3, p. 59]

Symmetric-key cryptography has two main ways of encrypting the plaintext, stream ciphers and block ciphers. Stream cipher is where a random stream of bits is taken, which will be the key, and then encrypting the plaintext by using a logical operation called "exclusive or" (XOR) on it. XORing means taking the plaintext and the random key, comparing each individual bit and if they are the same write down 1 and if they are different write down 0. Essentially this is how modern OTP works, as opposed to the historical OTP which used modulo arithmetic on alphabet letters. There are some issues of practicality that arise with stream ciphers. For example both parties have to have the same stream and keep up the state of the stream so both know which part of the stream to use for encryption and decryption without reusing old parts. This is called the synchronized mode. [3, p. 76-81.]

Another way called unsynchronized mode is where the stream is generated using a seed and an initial vector *IV*. The generation depends on a pseudorandom generator, that is out of scope for this study, but for the purpose of this explanation it is assumed that the pseudorandom generator is very strong at generating randomness. The seed is always kept secret and shared with the communicating parties. With every encryption an *IV* is chosen at random and sent to the recipient with the ciphertext. The recipient can now use the seed and the *IV* to generate the same stream and use it to decrypt the message. [3, p. 76-81.]

Block ciphers take the plaintext and slices it into fixed sized blocks, if a block is not filled completely by the plaintext it can be concatenated with the appropriate amount 0's to fill the block. Then a key is used to encrypt each block. There are many different designs and modes of operation of block ciphers. Simplest and most insecure one is called the Electronic Codebook (ECB) mode, where each block is encrypted and decrypted separately. This is insecure because equal plaintext blocks result in equal ciphertext blocks and patterns can be recognized. [3, p. 93-95.]

The most common one Cipher Block Chaining (CBC) mode works by generating an initial vector *IV* and XORing it to the first plaintext block before using the key to encrypt the block, then the resulting ciphertext is used as an *IV* on the next block in the same fashion (Figure 1). [3, p. 95; 4, p. 9.]

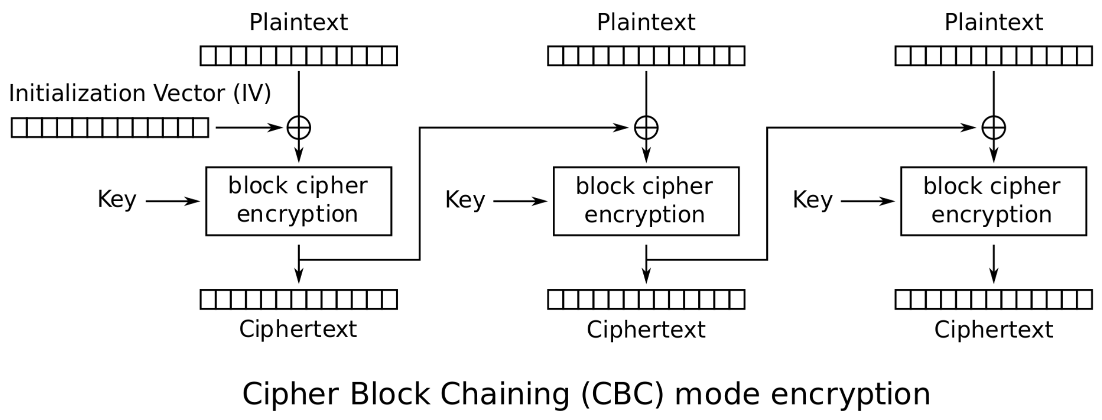


Figure 1: Cipher Block Chaining (CBC) diagram [33].

The *IV* is sent to the recipient with the ciphertext. This mode fixes the issue of ECB where equal plaintext blocks result in equal ciphertext blocks, as the *IV* is always generated randomly before each encryption. The drawback that is often discussed is that this mode cannot be parallelized since each block is dependant on the previous one during the encryption resulting in slower encryption speeds. However, in decryption each ciphertext block acts as an *IV* for the one ahead of it thus each block can be decrypted in parallel since they are not sequentially dependant on each other. Most notable block ciphers are Data Encryption Standard (DES), its big brother, Triple DES (DES-3) and Advanced Encryption Standard (AES). DES and DES-3 are mostly unused in modern applications these days, because of their short key size, but AES remains in wide use and remains uncracked. [3, p. 95; 4, p. 10]

2.2.1 Data Encryption Standard

Data Encryption Standard is defined as a symmetric-key block cipher and published by National Institute of Standards and Technology (NIST), formerly National Bureau of Standards (NBS). It started in 1972 as a request for a new national encryption scheme by the NBS that could be used commercially and could secure data in transit and at rest. IBM submitted one of their cryptographic algorithms designed for financial applications and after extensive review and further development by the National Security Agency (NSA) and the public. It was accepted as a Federal Infomation Processing Standard (FIPS) 46 in 1977. [5, p. 143; 6, p. 1.]

As underlying block cipher design, DES uses the Feistel network. Feistel network is a popular design of block ciphers used in many modern encryption schemes. In Feistel network a block of plaintext is separated into two halves and they go through rounds. Each round, some function *F* takes a sub-key derived from the master-key and one half of the block and encrypts it. This encrypted output is then XORed to the other half and then the halves are swapped and another round is done to it as can be seen in Figure 2. [6, p. 2.]

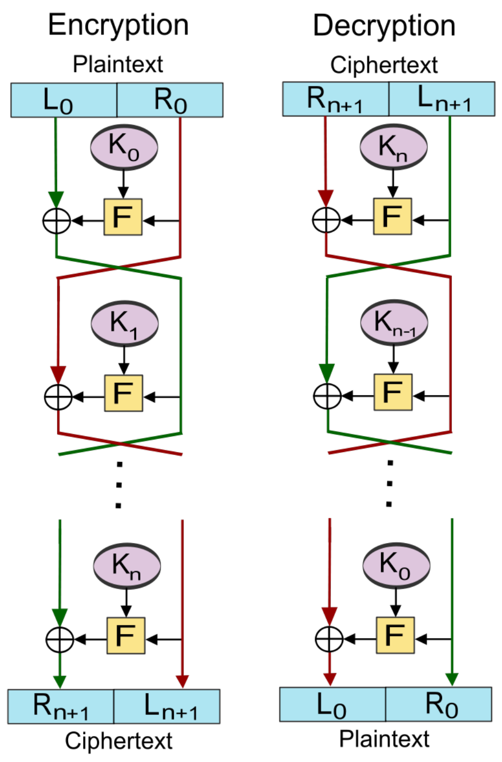


Figure 2: Diagram of Feistel network [32].

The function and sub-keys are not defined in Feistel network so it is up to the actual design of an encryption scheme to define the function and the derivation of sub-keys from the master-key. The derivation of the sub-key is called the key schedule. The function usually consist of so called substitution boxes (S-boxes) and permutations. [3. p. 160-162.]

DES is a 16 round Feistel network with a block size of 64 bits, key size of 56 bits and sub-key size of 48 bits. The encryption process is made of two permutations called P-boxes and the 16 Feistel network rounds in between. The P-boxes take 64 bit input and permutes them according to a predefined rule. As the P-boxes are predefined and public knowledge they have no effect on the security of the scheme and are ignored in the scope of this thesis. It is believed that P-boxes are in DES, only to slow down software implementations and attacks although no official explanation has been given from the DES developers. [5, p. 144-145; 3, p. 163-165.]

As can be seen in Figure 3 one round of DES takes the two halves of a block (R_I and L_I), puts the other one through a function f called DES function and XORs the output to the other half. Then the sides are swapped.

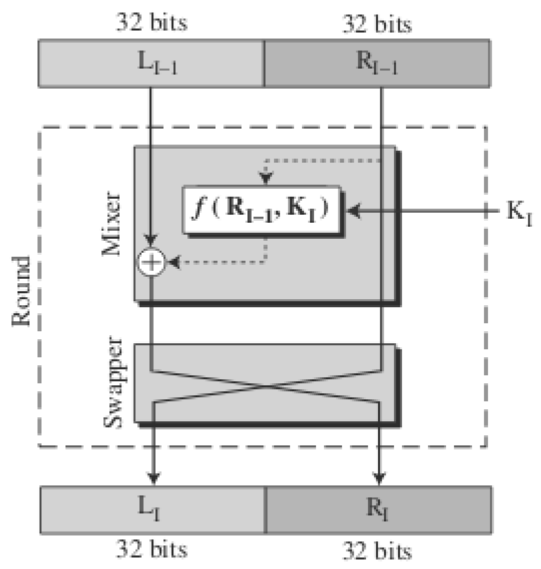


Figure 3: One round of Data Encryption Standard (DES) [5, p .146].

In Figure 4 DES function takes the input of 32 bits and puts it through an expansion D-box to make it 48 bits. It then XORs the output with the 48 bit sub-key and puts that output through S-boxes to make it back into 32 bits. That output is then put through another straight D-box and finally the output is XORed with the other half of the block. The sides are swapped and it goes through another round, in total of 16 times for each block. [3, p. 163-164; 5, p. 147.]

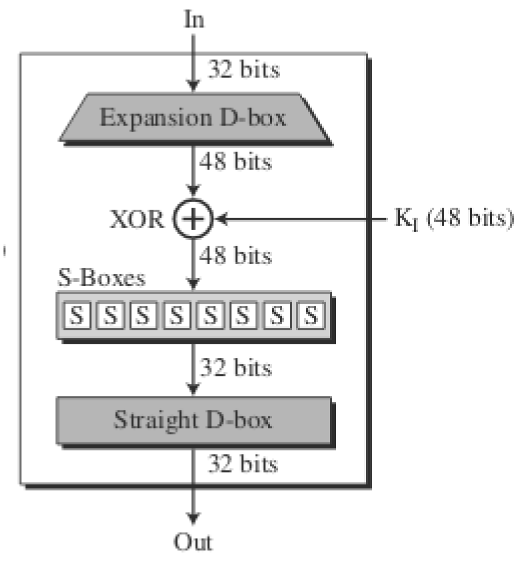


Figure 4: Diagram of the DES function [5, p. 147].

DES key size is not sufficient enough for modern times but the cryptography of the DES function has been tested and proven to withstand immense scrutiny from mathematicians trying to break it. The most succesful attack against DES is to bruteforce the key. An adversary only has to check 2^{56} keys to find a match and in 1998 a special computer was built that was able to do it in 112 hours. [3, p. 169-170.] Since the key size is the obvious problem and slightest change to the inner DES function, to fit a larger key, could make the whole design insecure, a double-DES (DES-2) was introduced. DES-2 goes through the DES twice as the name suggests and used two different 56 bit keys, increasing overall key size to 112. This did not yield enough of a security increase because of an clever mathematical attack called meet-in-the-middle, that could use the two rounds of the DES and the outputs to figure out the key pair in almost the same amount of time. Then the DES-3 was introduced that fixed the meet-in-the-middle attack and could use the 2 keys like in DES-2 or increase the key size to three keys, 168 bits. DES-3 was raised as the new standard but doing the already inefficient DES 3 times made it way too slow and another encryption scheme was needed. [3, p. 170-172.]

2.2.2 Advanced Encryption Standard

AES was again started as a request for a new block cipher by NIST in 1997 with security and efficiency as the most important criteria. It was decided to be a very public competition to amass most amount of scrutiny from the public cryptographers around the world. Up until this point most of the greatest cryptographers especially in the US worked for the military or the government. The competition was cleverly designed and it was split into 2 conferences, where the schemes would be evaluated thoroughly to have the best ones make it to next round and in the end only one would be deemed the best. This incentivized the developers of the encryption schemes to try and break each others schemes and find the slighest weaknesses in them. In the end algorithm called Rijndael designed by John Daemen and Vincent Rijmen from Belgium won the competition and it was deemed as the Advanced Encryption Standard by NIST in 2000. [3, p. 173; 6, p. 3; 7, p. 9-15.]

Rijndael differs from the standardized AES. Rijndael was designed to be a flexible block cipher that supports multiple different block and key sizes. AES however is fixed at block size of 128 bits and key size of 128, 192 or 256 bits. [7, p. 21.]

AES is essentially a substitution-permutation network instead of Feistel network, although they are very similar and both consists of multiple rounds. In substitution-permutation network in one round the input bits are sent through S-boxes where the bits are substituted in a certain way mandated by the designer of the scheme, after which the outputs are permutated. These two steps are to add confusion and diffusion, which prevent cryptanalysis attacks. A sub-key is usually XORed to the inputs before the substitution stage. [3, p. 174]

AES algorithm consists of either 10, 12 or 14 rounds, depending on the key size. Each round has 4 stages or transformations that are done to a so called state. In AES the state is a 4 by 4 array of bytes that is part of the plaintext. The sub-key or as called in AES, round-key, is the same size as the state. The 4 stages are as follows:

1. AddRoundKey: A 16 byte or 128 bit round-key is derived from the master-key and represented as 4 by 4 bytes. It is then simply XORed to the state.
2. SubBytes: Each byte of the state is transformed according to a fixed S-box.

3. ShiftRows: Each row is cyclically shifted by a certain amount depending on the number of the row. First row is not shifted at all, second row is shifted one step to the left, third is shifted two steps to the left and th fourth row is shifted three steps to the left.
4. MixColumns: Each column is multiplied by a fixed matrix.

In the final round MixColumns is replaced with an additional AddRoundKey stage. [3, p. 174; 7, p. 20-23.]

AES has seen a immense amount of scrutiny over the years and only known attacks against it rely on less rounds than optimal and even then the attacks are not practical and take some time. The only practical attack against AES is to try and use exhaustive search for the key which takes 2^{256} guesses, if a 256 bit key is used. This number is so astronomically large it is hard to even imagine. [3, p. 175]

2.3 Asymmetric-key Cryptography

Asymmetric-key cryptography also known as public-key cryptography aims to solve the key sharing problem of private-key cryptography. As described in the Chapter 2.2, private key cryptography requires the participants to have the same key. The distribution needed to happen in a secure way, such as meeting in person or already established encrypted channel for example, to stop adversaries from obtaining the secret key.

The basic concept of the public-key cryptography is to have a two keys mathematically related to each other, public key and a private key. The private key is always kept secret and the public key is shared. The messages are always encrypted with the recipients public key. Once encrypted it can only be decrypted by the associated private key only known to the recipient of the message. It is important that while knowing the public key it is impossible for the adversary to decrypt the message. This solves the key distribution problem as everyone can now share their public keys without a worry as long as their private keys are kept secure. [8.]

In 1976 Whitfield Diffie and Martin Hellmann published a revolutionary cryptography paper titled ”New Directions in Cryptography” that outlined many problems in private-key cryptography and introduced many ideas in the cryptography domain. They introduced the public-key cryptography, the Diffie-Helmann key exchange as well as message authentication codes or MACs [9, p. 1-5]. The key exchange is a way of generating secret keys for communicating parties over a public channel without knowing any shared secrets and the MAC is a way of signing a message and anyone with the corresponding public key is able to verify the authenticity of the message. Diffie and Hellmann realized that some mathematical functions are easy to calculate one way but doing it in reverse is extremely hard and takes a very long time [3, p. 306-308]. Such problems that are conjectured to be hard in math are prime factoring, discrete logarithm and elliptic curves [3, p. 231]. Some important algorithms that this chapter does go over are: RSA, Diffie-Helmann key exchange and Elliptic Curves.

2.3.1 RSA

In 1977 three scientists in Massachusetts Institute of Technology (MIT) developed an algorithm based on public-key cryptography work published year earlier by the Whitfield Diffie and Martin Hellmann, and named it after their last names, Ron Rivest, Adi Shamir and Leonard Adleman. British intelligence agency Government Communications Headquarters (GCHQ), had developed similar public key scheme but it was only declassified and published publicly in 1997. [8.]

RSA algorithm is based on a seemingly very simple mathematical problem that has been studied for hundreds of years and no efficient solution has been found yet, the factorization problem. The factorization problem is the problem of finding integers *p* and *q* for a given non-prime integer *N* in such a way that $pq = N$. It is a computationally hard problem to solve. One way to solve the problem is to exhaustively search whether *p* divides *N* by checking each integer one by one until $\sqrt{N}$ because the smallest prime factor of *N* can only be as large as $\sqrt{N}$. Although more efficient algorithms have been found none are very efficient. The problem gets very time consuming when *p* and *q* are prime numbers and extremely large. [3, p. 248-249.]

RSA algorithm expands on this problem by using modulo arithmetic and the factorization to create the reasonably efficient encryption method. The in-depth mathematical functions for key generation are not in the scope of this thesis. In principle, three very large numbers are generated. *N* used as a modulus is chosen by picking two random primes *p* and *q,* where *pq* = *N.* Also integers *e* and *d* are generated related to *N.* The public key is $\langle N,e\rangle$ and the private key is $\langle N,d\rangle$. The ciphertext *c* is generated from a plaintext message *m* in a following way: $c=m^e \mathit{mod}\, N$. Similarly the decryption is done by: $m=c^d \mathit{mod}\, N$. Even if the adversary would be able to find every piece of information except the integer *d,* it is extremely hard to computate the integer *d* which essentially is the private key. [3, p. 258-259.]

2.3.2 Diffie-Helmann Key Exchange

Diffie-Helmann key exchange relies on the same factorization problem utilized by the RSA algorithm. Diffie-Helmann key exchange is a way for two parties to generate a shared secret key over open network. Meaning that even if an eavesdropper can listen to everything between the two parties the eavesdropper cannot derive the same shared key. After the shared key is generated for both parties they can start communicating using symmetric encryption of their choice.

The key exchange is done by choosing a large prime *p* and a generator *g,* such that *g* is a primitive root modulo *p.* These two numbers are agreed upon by both parties, lets call them Alice and Bob. Alice then chooses a secret integer *a* and Bob chooses a secret integer *b.* Alice computes $A=g^a \mathit{mod}\, p$ and Bob computes $B=g^b \mathit{mod}\, p$. They then exchange the computed values *A* and *B.* With the knowledge of *B* Alice can compute $K_1=B^a \mathit{mod}\, p$ and with the knowledge of *A* Bob can compute similarly $K_2=A^b \mathit{mod}\, p$. The computed values K_1 and K_2 are equal because in modulo *p* the following is true: $A^b \mathit{mod}\, p=g^{ab} \mathit{mod}\, p=g^{ba} \mathit{mod}\, p=B^a \mathit{mod}\, p$. This allows two people to calculate exact same secret key without anyone listening in on the exchange being able to. [11, p. 10-12.]

3 Matrix Open Standard

Matrix is an open standard for communications over IP. It can be used for instant messaging, VoIP (Voice over IP) calls over WebRTC protocol or for any kind of application that needs to send and receive messages over IP and retain the conversation history. It aspires to be a decentralized communications network that would allow everyone to converse together without worrying about what application they are using or without having a central set of servers that collect everything. It wants to break the current communications silos and be universal and as easy to use as email. Matrix itself is not an application, instead it is a standard that defines how it works and it also provides open source reference implementations of Matrix servers, client Software Development Kits (SDK) and some application services. The communication to the server is done through HyperText Transfer Protocol (HTTP) Application Programming Interfaces (API). HTTP is the protocol that all of the internet works on and APIs are often used in servers to receive data and respond to data queries. This allows Matrix to be easy to implement on the client level. [12.]

3.2 History

Matrix was first created in 2014 inside a company called Amdocs and was called Amdocs Unified Communications [14]. The protocol was funded my Amdocs and developed by a small team lead by Matthew Hodgson. In 2015 Amdocs created a new subsidiary called ”Vector Creations Limited” where the team developing Matrix was then moved. In 2017 Amdocs was stopping the funding for the team which lead to them creating their own company called ”New Vector” based in the UK. This new company was in charge of developing and maintaining the Matrix standard and hosting the public Matrix servers. It also provided consultancy and hosting services on top of the Matrix protocol. The company was not well funded and relied on donations to keep the public servers running. New Vector received its first big funding round in January 2018 from Status, an Ethereum cryptocurrency based start-up, of 5 million US dollars [15]. In April of the same year the French goverment announced a move to start using Matrix protocol and the Riot application developed by ”New Vector” for all their governmental communications in order to protect them from US or Russian spying [16; 17].

In October 2019 Matrix.org foundation was created in order to keep Matrix as open and independent as possible without compromising it through commercial interests. It is founded in the UK as a private company limited by guarantee and as a community interest company to make it a non-profit organisation with an asset lock to prevent the selling of the intellectual property or extracting of profits [17].

Very recently in 2019 the New Vector raised a Series A funding round of 8.5 million US dollars from Notion Capital, Dawn Capital and Firstminute Capital. The funds are ment to go in to the development of Matrix standard in form of richer End-to-End encryption support and loads of other features. For the New Vector the funding is used for their messaging app Riot and their Matrix hosting platform Modular.IM. The investment is said to be impartial on the development of the Matrix standard and the commercial gain for the investors will come from the commercial platform New Vector is building on top of the Matrix standard, like the hosting service of Modular.IM. [18; 19.]

3.3 Architecture of Matrix

Matrix consists mainly of homeservers and clients. There are also identity servers and application services but first this thesis goes over the more important parts, homeservers and clients. Homeservers are Matrix servers that clients connect to. Clients are the applications that users use, so in practise clients are users or devices. Anyone can host their own homeserver and only have themselves registered on it, or anyone can use a larger homeserver such as Matrix.org homeserver that has hundreds of thousands of users. Clients do not have to be in the same homeserver in order to communicate Inside these homeservers are something called rooms. They are a virtual environment where events such as messages get sent between the clients participating in the room. Rooms can be between two clients or thousands of clients. The decentralization of the Matrix comes from the fact that each room is stored on every single homeserver participating in the same room. This creates a federated network of homeservers and clients where no central server exists as can be seen in Figure 5.

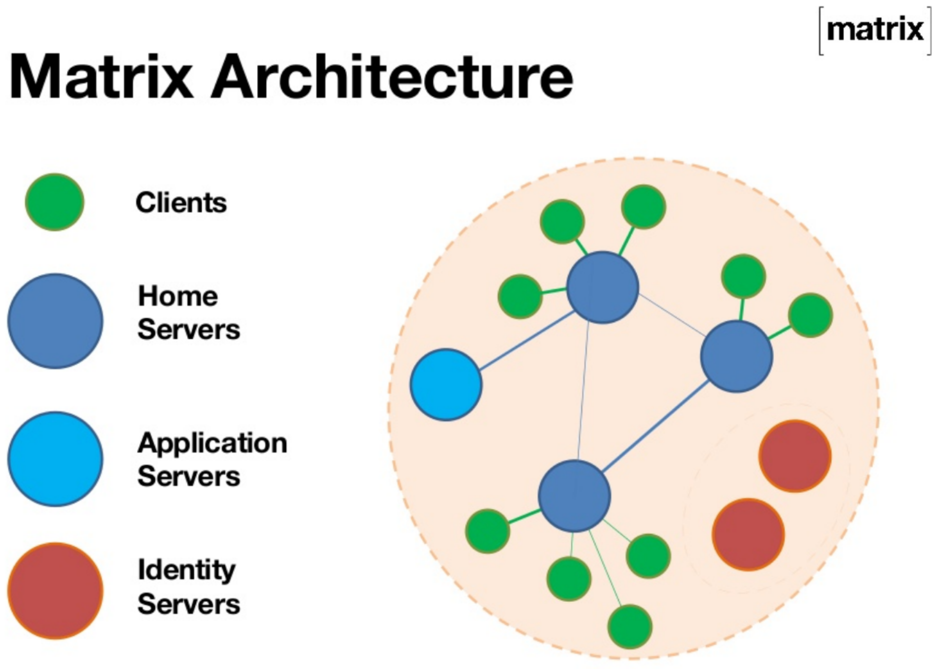


Figure 5: Architecture of Matrix ecosystem [34].

A homeserver can be kept private by disabling the federation feature. This allows for private homeservers and even private federations of homeservers. For example the French government will equip every branch and office of the government to have their own homeserver and only allow connections from whitelisted list of other homeservers in the French government. This means is that each office of the government can have private rooms between the people in their office or collaborate with other offices by inviting users from different branches, thus creating decentralized rooms. As the data is shared between homeservers every participating homeserver shares the same conversation history, minimizing data forgeries and enabling transparency between homeservers. [20.]

Matrix identities are usually in the form of ”username@homeserver.com”. Usually when creating a Matrix account an username is chosen and additional information can be provided such as email or phone number for password resets and other additional control over the account. Identity servers job is to map these ”3rd party identifiers”

(3PID) to the Matrix identities for users to discover other users through their email or phone number. Identity servers and the 3PIDs are optional and not a requirement in the Matrix ecosystem. Matrix.org homeserver uses identinty servers by default to make it easier for users to find their friends in the Matrix. [20.]

As stated above briefly, Matrix uses HTTP APIs for the server communications. Essentially Matrix is just a detailed list of specifications (spec) for these open APIs that anyone can implement on their server or on their client. Matrix.org has an open source reference homeserver that implements these APIs, called Synapse. The APIs themselves are common APIs that are used everywhere, JSON over REST. REST or Representational State Transfer is a commonly used software architecture that defines how a web service behaves. JSON is Javascript Object Notation and it is used to transmit human-readable data objects.

Matrix has three different APIs that creates the whole standard. Theres client-server API, server-server API and application service API. Client-server API defines endpoints for the client to get or send data to the server such as messages. Server-server API defines the interaction between homeservers [20]. Application Service API defines extra functions on the server that a normal client would not be able to do. The application service API was created to give some services more powers, and allow bridging to other application and protocols. Bridging, as the name states, is a bridge from another application to Matrix. Messages sent in a Matrix room are also sent to the bridged application which could be Telegram for example. [21.]

The basic flow of sending a message from client A to client B when both are on different homeservers can be seen in Figure 6. Client A sends HTTP PUT request to its homeserver using the client-server API. The homeserver then signs the message and appends it to its copy of the room events. It then sends the message to client B's homeserver as HTTP PUT request using the server-server API. Then client B's homeserver authenticates and verifies the signature and appends the message into its copy of the rooms events. Client B can then get the message via HTTP GET request. [20.]

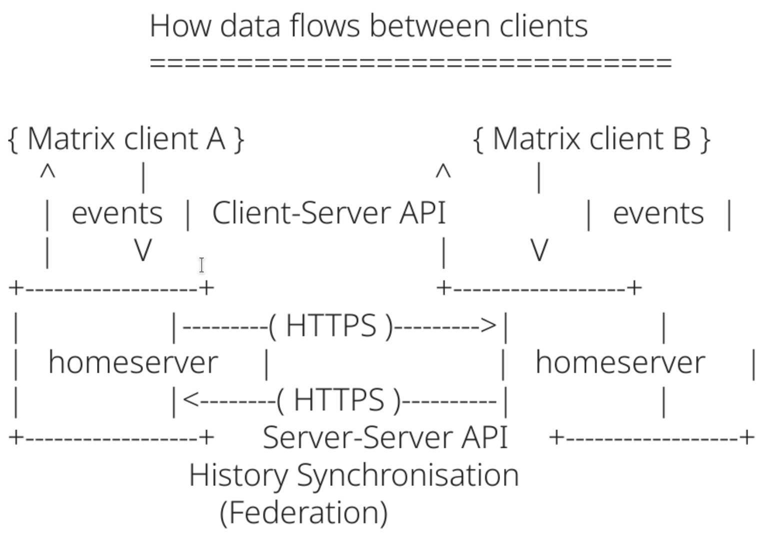


Figure 6: Message flow of between two users from two different homeservers [20].

In the most basic form messages are extremely easy to send in Matrix. The HTTP request in Listing 1 is extremely simplistic with only two lines of JSON in it: what type of message it is and the contents of the message. Of course often the user is authenticated and the events can be more complex which would add more things in to the request but it's a good demonstration how easy it can be to create simple Matrix client using these APIs.

```
PUT
/_matrix/client/r0/rooms/%21636q39766251%3Aexample.com/send/m.room.messag
e/35 HTTP/1.1
Content-Type: application/json

{
  "msgtype": "m.text",
  "body": "hello"
}
```

Listing 1. Simple PUT request for sending a message in Matrix [22.].

As people in the modern day can and often do have multiple different devices they use in their lives such as laptops, desktops, smartphones and tablets. The way Matrix spec defines these ”devices” does not always relate to only a physical device, it can also relate to different browsers on a computer or different Matrix clients on a smartphone. Devices are usually used for handling encryption keys as each device have their own encryption keys but users can also revoke access on some devices in case of theft for example. [20.]

Matrix is event oriented as briefly touched above. Meaning that everything exchanged over Matrix is called an event. Events are usually sent in rooms between users. Events can be simple things like basic messages or they can be more complex ones that are not even visible to the user like iniating encryption and sending session identifiers and encryption keys to the device as a ”to_device” event. Matrix spec has many different events and different clients can also create their own unique events. Naming conventions for clients own events uses Java’s package naming conventions for example ”com.example.myapp.event”. Events detailed in the Matrix spec use the namespace ”m.” for example ”m.room.message”. [20.]

3.4 Encryption in Matrix

Matrix uses similar cryptographic schemes as Signal, probably the most known secure communications application on the market. The parent organisation that created Signal, Open Whisper Systems, created an open standard for secure communications called the Signal protocol. The Signal protocol can be used to provide end-to-end encryption for voice calls, video calls and messages. For the Signal protocol they designed Double Ratchet Algorithm which is an algorithm for key management. They also designed an improvement on Diffie-Helmann Key Exchange called the extended triple Diffie-Helmann (X3DH).

Matrix took the Double Ratchet Algorithm and altered it slightly to be more suitable for group chats with hundreds or thousands of users. This resulted in two different algorithms that Matrix uses for one-to-one messaging and group messaging called, ”Olm” and ”Megolm”.

3.4.1 Double Ratchet Algorithm

Double Ratchet Algorithm as its name states, is not a mathematical cryptographic scheme, it is an algorithm for key management for encrypted sessions. First in Double Ratchet Algorithm the parties need to agree on a shared secret key through Diffie-Helmann Key Exchange or the variation of it the X3DH. After the shared secret key is established the parties can start conversing using the algorithm. The algorithm derives new keys for each message in a way that prevents the earlier keys being calculated from the current or future ones. It also uses Diffie-Helmann calculations as part of each message to prevent future keys being calculated from current or past ones. [23, p. 3.]

There are three main concepts in the algorithm that combined create the algorithm. There is Key Derivation Function (KDF) chains, Symmetric-key ratchet and Diffie-Helmann ratchet. [23, p. 3.]

KDF is defined to be a cryptographic function that takes a secret KDF key and input and spits out output data. The chain comes from chaining these KDFs together using some of the output data as the KDF key for the next KDF as can be seen in Figure 7. Which demonstrates the three rounds of KDF chain resulting in three output keys.

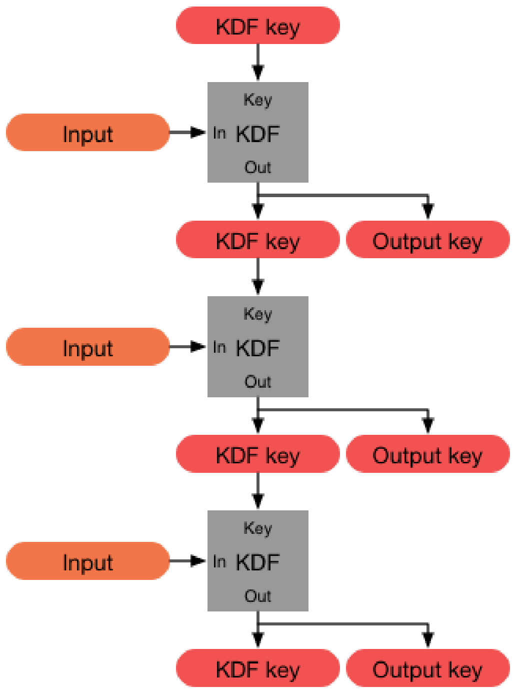


Figure 7: The Key Derivation Function (KDF) chain [23, p. 4].

In a Double Ratchet session between two parties, both parties need to have three of these chains, a root chain, a sending chain and a receiving chain and have KDF key for each of the chains. The sending chain of one party matches the receiving chain of the other party. [23, p. 4.]

The symmetric-key ratchet ensures that every message sent between the parties is encrypted with a unique message key. These message keys are outputs from the sending and receiving KDF chains and the KDF keys for these chains are referred to as chain keys. In each round the KDF outputs a chain key and a message key and the chain key is used for the next round of the ratchet while the message key is used for the encryption. In Figure 8 it is demonstrated how the symmetric-key ratchet works by showing two rounds of a sending or receiving chain. [23, p. 5.]

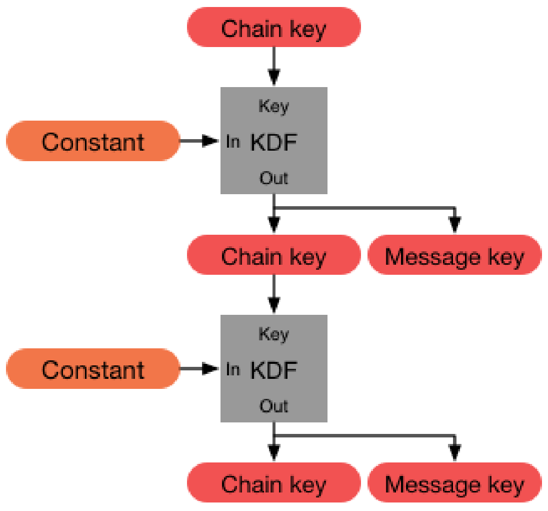


Figure 8: Symmetric-key ratchet [23, p. 5].

Since the inputs are constants if an attacker would get a hold of one of the parties sending and receiving chain keys they could calculate all future message keys and decrypt all future messages. For this reason Diffie-Helmann ratchet is added to the mix. [23, p. 5.]

Both parties will generate Diffie-Helmann (DH) key pairs, public and private keys, and the public key is sent as part of a message. When a new DH public key is received a DH ratchet step is calculated which calculates new DH key pair for the party. As a result each party is always sending new DH public keys as part of a message and calculating a new DH key pairs for themselves. This protects future messages. If an attacker managed to get the private DH key from one of the parties it would not matter as new DH key pair will be calculated most likely in the next few messages. [23, p. 6.]

Figure 9 shows how the DH ratchet works when Alice and Bob send messages between each other. The figure only shows the DH ratchet and not the message handling. Both Bob and Alice start with a DH key pair and when Bob sends his public key to Alice, Alice calculates a DH output with her private key and Bobs public key. When Alice sends her public key to Bob, Bob can calculate the same DH output using his private key and Alices public key. Bob also generates new DH key pair and uses Alices public key and his new private key to calculate new DH output. Next time Bob would send a message he would send Alice his new public key and Alice would do the same steps as Bob just did. This exchanging of keys and calculating new pairs goes on indefinitely. [23, p. 6-11.]

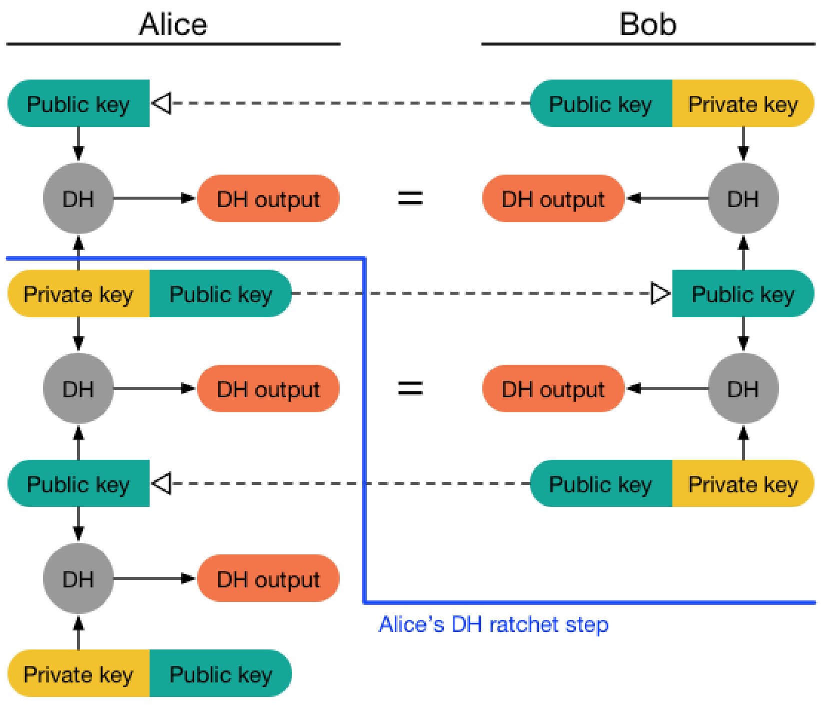


Figure 9: The Diffie-Helmann Ratchet [23, p. 9].

The Double Ratchet name comes from combining these two ratchets together. In Figure 10 it is demonstrated how these ratchets are implemented together. The DH output is used as an input for KDF in the root chain which generates receiving and sending chain keys.

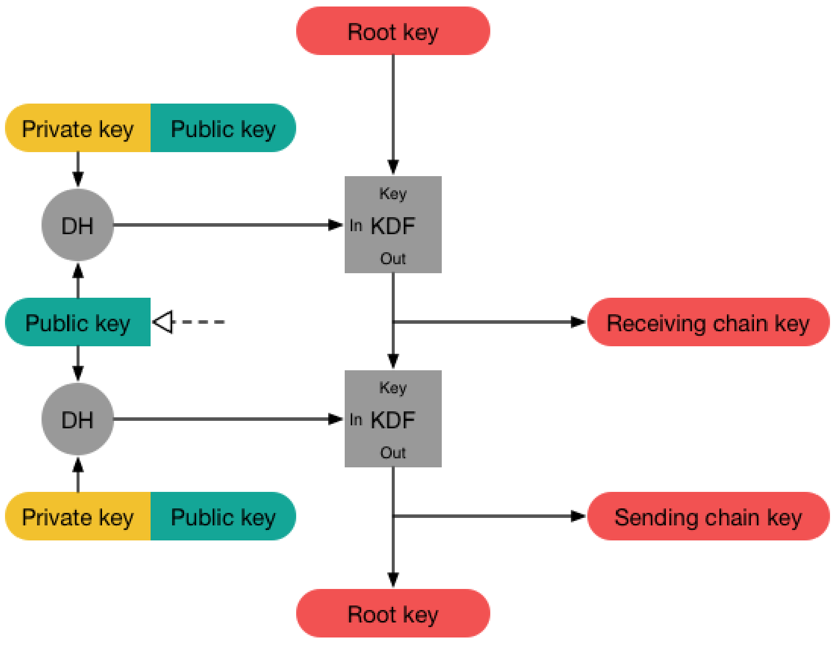


Figure 10: Diffie-Helmann ratchet and Symmetric-key ratchet combined together [23, p. 12].

So when a message is sent and a new DH public key is received a DH output is calculated using current DH key pair and fed into the root chain KDF and out comes the next root key and a receiving chain key. Then a new DH key pair is generated and used to calculate second DH output that is then fed into root chain KDF and out comes the next root key and a sending chain key. This way Bobs sending chain matches Alices receiving chain and both can calculate the same message key to decrypt the message and vice versa. [23, p. 12-16]

3.4.2 Olm

Olm is a slightly different implementation of the Double Ratchet Algorithm and is used to establish a secure session between parties in Matrix and used to share keys for Megolm session. It can be also used for one-to-one messages but Matrix recommends that it is only used to share Megolm keys between parties. [24]

In Matrix each device has their own identity key pair and multiple one-time key pairs that are generated using the Curve25519 elliptic curve. Each device then publishes their public parts of identity keys and public parts of one-time keys to the server. [24]

The initial setup for the Olm requires four Curve25519 inputs, identity keys for both parties and, and one-time keys for both parties. Both parties generate a shared secret using the X3DH. Then from this shared secret both parties derive a root key and a chain key using a KDF. Here is where Olm implements Double Ratchet Algorithm a little bit differently. Instead of having three KDF chains they only have two because the sending and receiving chains are combined to one. However in the sending/receiving chain the message keys are numbered in the order they are created and it is agreed that the other party uses the even numbered ones and the other odd ones for encryption and vice versa for decryption. After the session is initiated and both parties have calculated the secret key they can start conversing using the Double Ratchet Algorithm. [24]

The actual encryption used in Olm is AES-256 in Cipher Block Chaining mode as described in Chapter 2.2.2. The AES-256 key as well as an authentication key used to authenticate the sender are derived from the message key that was generated by the sending/receiving KDF chain. [24]

3.4.3 Megolm

The Double Ratchet Algorithm is very resource intensive for group messaging since each person in the group chat would have their own Double Ratchet session with each user. This would result in huge amount of keys being generated and tracked and huge amounts of encryption the processor has to do since each message has to be encrypted for each participant in the group chat separately. Here comes Megolm which is designed to support group messaging in group consisting of thousands or even tens of thousands of participants. It is described as an AES-based cryptographic ratchet. [25]

Megolm sessions consists of a counter, Ed25519 key pair and a ratchet. The Ed25519 is a digital signature algorithm based on elliptic curves and is used for authentication in Megolm. The ratchet is just 1024 bits of randomly generated data. These three fields form the session data. The session data is shared between each party by first establishing a secure Olm session with each participant and sending them the session data encrypted with the Olm session. Each participant in the conversation has to generate their own Megolm session and share it with everyone else individually. Each time a message is sent it is encrypted using the participants own Megolm session and it is decrypted by receiving participants by using the same session that was shared to them earlier. To ensure new keys are used for each message, the ratchet is forwarded using the Megolm ratchet algorithm. [25]

In Megolms case if an attacker gains access to the session data they can decrypt any future messages sent by the victim. This is a known limitation in Megolm and can be fixed by forcing the creation of new Megolm sessions after a certain number of messages or after a new participant joins a group chat. [25]

4 Qt Development Framework

Qt is a application development framework intended to use for building cross-platform applications. Qt applications can be run on nearly every operating system and platform, for example: Linux, OS X, Windows, VxWorks, QNX, Android, iOS and BlackBerry to name a few. Qt is a framework written in C++. It extends C++ by adding own helpers, widgets and functions to it and compiles them in to standard C++ after which the C++ can be compiled with any C++ compiler like GCC for example. This is what makes Qt so powerful and why it can run on pretty much every platform. It was also used by Nokia in their old phone operating systems like Symbian and MeeGo. [26; 27.]

One of the more powerful extensions Qt provides is the signal and slot system. If Java and design patterns are familiar it would be closest to the observer design pattern.

Essentially it connects slots to signals. Slots can be functions, so when a signal that it is connected to fires, the slot gets executed. It is most often used to signal that something changed within a class so other classes with slots for that signal can act accordingly. [35]

Qt is not just C++, it also offers a language called QML to build Graphical User Interfaces (GUI) in a very intuitive and easy way. QML integrates Javascript for added functionality. QML could be compared to HTML+CSS, the web design languages, but has some benefits that make it more intuitive to use. Most applications built with Qt have a frontend built with QML and all the backend functionality built with C++. Qt also provides standardized components called Qt Quick Components to build applications that easily scale to different screen sizes and that look universal. [26.]

Qt provides their own Integrated Development Environment (IDE) called the Qt Creator. It has all the features most IDEs have such as: intelligent code completion, syntax highlighting, debugger and integration to most version controls like git. Theres also a drag-and-drop type design tool called Qt Designer. [26.]

5 Implementing Encryption for Matrix Client

Once more theoretical knowledge is acquired about encryption and the caveats one might face, it is easier to start designing how the encryption should be implemented. The messaging application upon which the encryption is implemented, is a messaging app designed to run on any platform but mainly built to run on Necuno Solutions own secure mobile operating system. Currently it is minimalistic and only has necessary functions implemented using the Matrix open standard. In its current state it pretty much only allows messaging for one-to-one and group chats. The application is run on QEMU virtual emulator that emulates the customers own operating system. Figure 11 shows the QEMU window and the operating system.

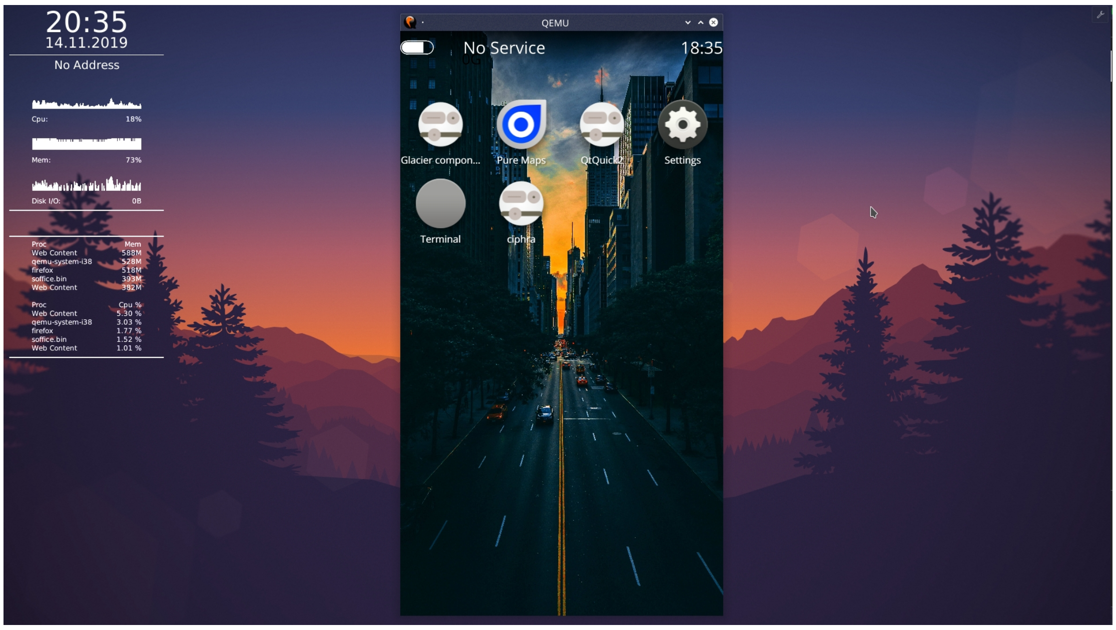


Figure 11: QEMU virtual emulator window on a Linux desktop.

5.2 Goals

Goals for the thesis is to get familiarized with cryptography, take a deep dive in to the most common algorithms and schemes, understand the Matrix Open Standard and design and implement encryption for a messaging application that communicates over Matrix Open Standard. The encryption for the messages needs to be up to standards and secure enough to be used in extreme cases such as in governmental communications for example. The implementation needs to be done in a way which allows the underlying encryption scheme or libraries to be changed with minimal effort.

In order to evaluate encryption schemes and algorithms some level of cryptography understanding is required.

5.3 Deciding on Encryption

Careful evaluation was done on which encryption scheme would be used. There are caveats that need to be considered when deciding encryption schemes. It is most often considered foolish to create one's own encryption schemes unless they are mathematicians and have very deep understanding of cryptography. Cryptography is so complex that trying to invent new schemes by mixing and matching or layering commonly used schemes does not necessarily make it more secure. So it is best to stick with proven schemes. Good encryption algorithm should be based on proven math and as simple as possible algorithm.

At this time there is a threat to encryption in form of quantum computing. Quantum computing has the possibility of breaking almost all encryption by making it trivial to calculate primes in the RSA problem by using something called Shor's algorithm. However for this thesis quantum computing is considered as a threat but not in the near future. So called ”post-quantum” encryption schemes will not be considered.

What needs to be considered for end-to-end encryption?

- Messages need to be strongly encrypted.
- Messages need strong forward and backward secrecy, so no future or past messages can be decrypted if one key is compromised.
- Sender should be able to be authenticated.
- The encryption should be efficient.

After considering all these basic requirements there aren't many options to choose from. The most used one that was discussed in Chapter 3.4.1 is the Double Ratchet Algorithm. It is extremely widely used in multiple different secure chat applications and it is hailed as the most secure communications encryption algorithm by many security experts. Another contender that was found to have similar properties was NaCl (or often known as libsodium). It is an encryption algorithm designed for secure communications. Its biggest drawback is that it lacks wide spread use.

When looking at the actual encryption the scheme should be AES or stronger as AES has not yet been cracked [36]. AES is also a standard that has been looked at for almost 20 years by experts. There have been scares of it being cracked but usually the vulnerabilities are only possible under specific conditions like using AES with less than 14 rounds or having keys that are related to each other in a special way.

Alternatives are risky because they do not have same amount of scrutiny. For example NaCl does offer a chance to use stream cipher called Salsa20 [37] which has been said to be extremely strong like AES, but it does not offer the same amount of safety as 20 years of mathematical scrutiny. There is also a very similar block cipher to AES developed by the Soviet Union that is now the Russian standard for encryption called GOST. It is almost identical to AES but it also lacks more scrutiny from the public and does not have widespread use.

The Double Ratchet Algorithm has also had some criticism about its creator and about the funding of Open Whisper Systems. For example Yasha Levine, an investigative journalist, takes a deeper look into Signal, Open Whisper Systems and its creator Moxie Marlinspike. Open Whisper System is a non-profit organisation that tries to create privacy and security tools for the public, but as Levine describes in his book it actually might not be that innocent. Marlinspike has been working for the CIA (Central Intelligence Agency) before he started Open Whisper Systems and most of the funding Open Whisper Systems receive is from the US government. [28, p. 257-258.]

Even while knowing the above and discussing it with the company it was deemed secure enough with its wide adoption and the fact that no one has yet found any glaring issues with the algorithm. So out of all the options the Double Ratchet Algorithm was chosen to be used for this thesis.

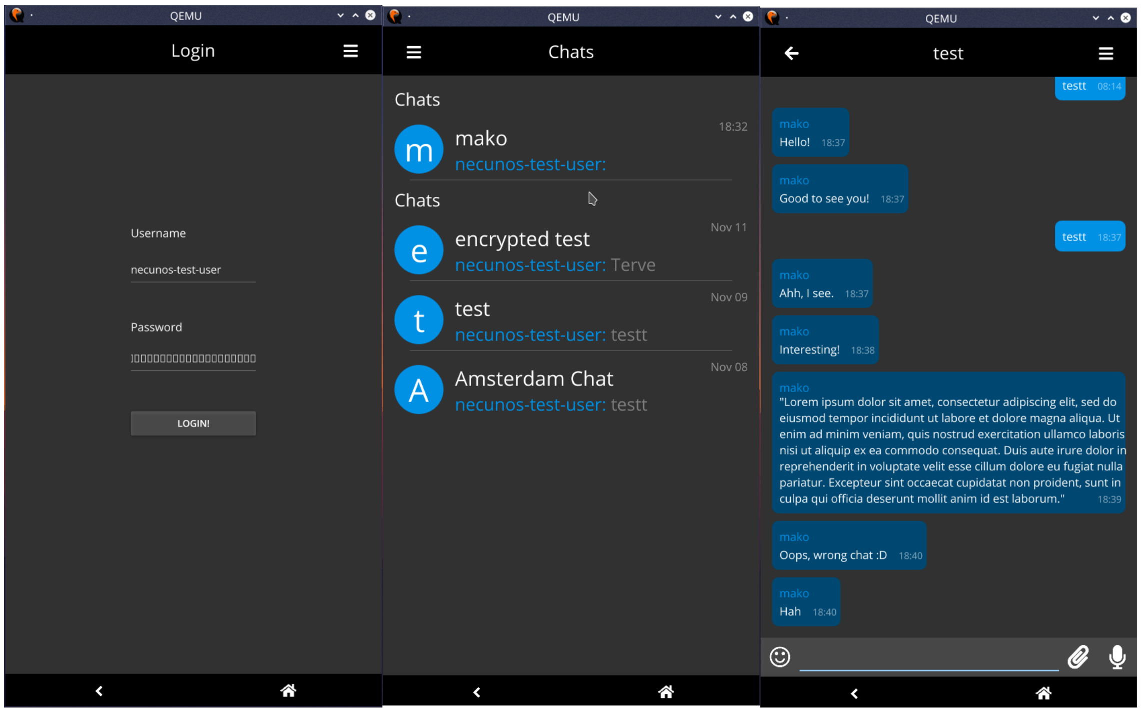
As stated before Double Ratchet Algorithm does not mandate which encryption schemes to use for the encryption, it is left for the implementations of the Double Ratchet Algorithm to decide. The most obvious implementation candidate is the Olm and also Megolm algorithms created by the Matrix team. Another open-source implementation is called OMEMO, which was created for another messaging protocol called XMPP. After careful consideration the Olm and Megolm took the cake. The main reason being that there has been some work done on Qt side to provide easy to use libraries for Olm and Megolm. These libraries are discussed briefly later on in the thesis.

So in the end it was the Double Ratchet implementation by Matrix that was chosen to be the encryption scheme. Good thing about Olm and Megolm is that they both use

AES-256 with CBC mode which is probably the most common and arguably most secure AES variation.

5.4 Application Architecture

The application that the encryption was for, was fairly simple and did require quite a lot of effort to understand the codebase. It is built on Qt and has few GUI pages done with QML: LoginPage, RegisterPage, MainPage and ChatPage. These are essentially the different views the application has and the UI works as a stack based architecture. For example when the app is started, first thing that pops up is the LoginPage, once logged in, the app replaces the LoginPage with the MainPage. MainPage lists all the rooms that are available to the user, so essentially the ones that the user has created or has been invited to. From there, one of the rooms can be clicked and the app will display the ChatPage which lists all the room events in chronological order just like the Matrix spec states. The three main pages can be seen in Figure 12.



Username
necunos-test-user
Password
)oooooooooooooooooooo

Chats
mako 18:32
necunos-test-user:
Chats
encrypted test Nov 11
necunos-test-user: Terve
test Nov 09
necunos-test-user: testt
Amsterdam Chat Nov 08
necunos-test-user: testt

testt 08:14
mako
Hello! 18:37
mako
Good to see you! 18:37
testt 18:37
mako
Ahh, I see. 18:37
mako
Interesting! 18:38
mako
"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum." 18:39
mako
Oops, wrong chat :D 18:40
mako
Hah 18:40

Figure 12: Three main views of the application.

The application handles all the backend stuff on C++ side where it has a *main.cpp* file which starts the application, then it has a C++ class called Connector which implements an interface called IConnector. The Connector class handles all the function calls between the user interface (UI) and the backend server. The reason why it implements an interface is so that there could be multiple different Connector classes without the need to change the UI. For example there could be a Connector for Matrix, a Connector for XMPP servers and a Connector to a completely new messaging service. All these Connectors provide the same functions for the UI so the UI doesn't have to be recoded everytime as long as the Connector implements the Connector interface.

The way message and room lists are handled, is through something called QAbstractListModel. It is an model interface provided by the Qt framework, that helps to handle lists with complex objects in it. The data can be changed in these lists very easily with the help of some predefined functions for handling updates and changes. The QAbstractListModel also helps to notify the UI when changes happen inside the lists. The application uses QAbstractListModel to handle events and rooms with classes called: MessageEventModel and RoomListModel.

The application also utilizes an open source Qt library called libQuotient (used to be called libQtMatrixClient) which handles all the connections to the Matrix server. It basically does a lot of work in the background, like handling sync messages with the server, it creates the HTTP requests and JSON payloads automatically. Only thing that needs to be done is to call a specific function and give it a few parameters and the library transforms it into a HTTP request and delivers it to the correct API end point. [30.]

In Figure 13 is a simple UML diagram of the backend side of the software without looking into libQuotient. The diagram shows how the Controller implementing the IController and QObject (necessary for it to use Qt's extra functions) is in charge of everything. The UI part also needs to see the RoomListModel and the MessageEventModels since they have to be directly linked to the UI through the Qt's modeling system.

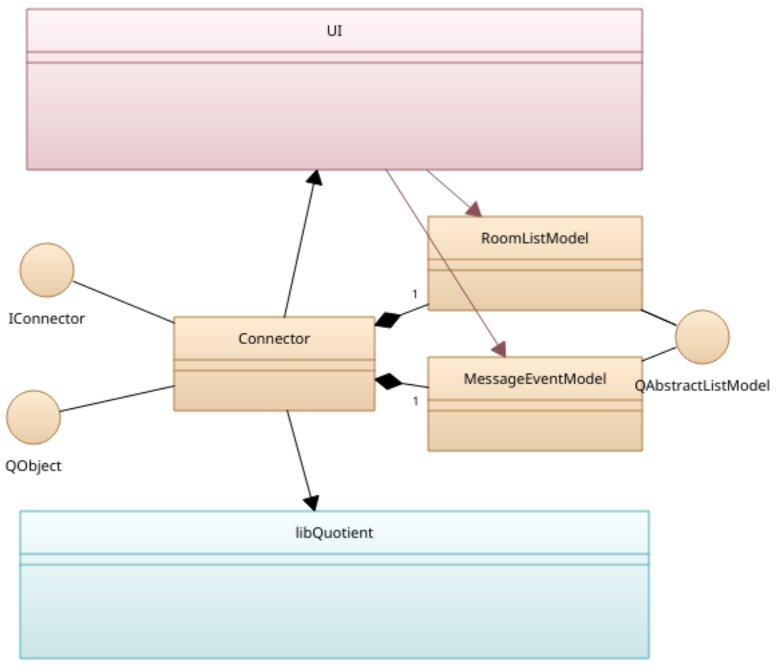


Figure 13: UML diagram of the application.

The MessageEventModel and RoomListModel are interesting since they use Qt’s signalling system to be up to date when data changes. In this specific code libQuotient provides signals when something changes such as *unreadMessagesChanged()* which can be basically subscribed to and when that signal is triggered all the subscribed functions, that are called slots, will be triggered. This signalling system allows MessageEventModel and RoomListModel to update themselves whenever there is a change in any room.

When designing encryption to be part of this system with fairly easy way of changing it, it was decided to be best to follow similar design approach as with Connector and UI. This would require an interface with the necessary functions to encrypt and decrypt messages. This results in a easy way of making new encryption classes that could provide different types of encryption without needing to change any other part of the application. For example a Connector that connects to Matrix could use a NaCl encryption protocol and send the encrypted payload over to the Matrix server.

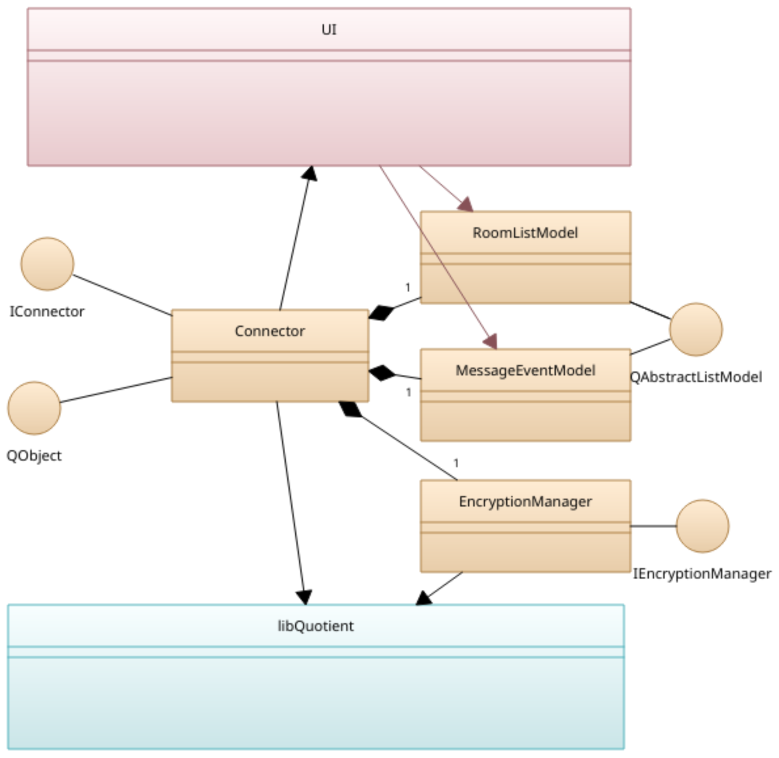


Figure 14: UML diagram with EncryptionManager Class.

Figure 14 shows how the new IEncryptionManager interface and EncryptionManager class implementing that interface fit to the UML diagram. In this case unfortunately the EncryptionManager class needs to know some necessary classes in the libQuotient so the final implementation is not as clean as it could be if the libQuotient library wouldn’t be used.

5.5 Implementation of Encryption

LibQuotient had started implementing encryption for the the library in the summer of 2019 which was right around the period when this thesis was worked on. There is also a QtOlm library created in early 2019 which provides basic error handling and easier to use functions for using the actual Olm library provided by the Matrix team [31]. For the libQuotient some work was already done on the library for decrypting messages but it was still a work in progress that only two people were doing in their free time. The library was completely missing encryption for messages and some parts of the key handling were improperly done. As the decryption was already partially done it was best to start with that. For testing the decryption, Riot.im, the Matrix flagship messaging application, was used since it already had Matrix encryption implemented.

The actual encryption and double ratcheting is done using the C/C++ Olm library that can be installed by cloning the repo and building it or on Linux for example some package managers have it already as a prebuilt package. The Olm library provides the primitive functions for key generation, decryption, encryption and session handling. QtOlm library just builds up on it with Qt based error handling and some helpers to make using the Olm more easier.

In the end it is a kind of an odd setup where our application is calling a library which then calls another library which then calls another library. All of these libraries are open source and each of them needed to be studied in order to make the whole setup work.

5.5.1 Decryption

The decryption starts with the MessageEventModel. First MessageEventModel has a *data()* function that gets called through the signalling system when an Event changes or new Event is added. This function gets the Event object and it is first checked if that Event is already known or if it is a brand new Event. If it’s a new Event its type is checked in order to determine how to handle it. So in this function the event can be checked if its an EncryptedEvent, a subclass of Event that is provided by the libQuotient. If it is EncryptedEvent the decryption should be tried by calling the *decryptMessage()* function in Connector (Listing 2). To keep the code clean and keep the MessageEventModel unaware of the EncryptionManager as designed, the Connector will then call the EncryptionManagers *decryptMessage()*, which then calls the libQuotients *decryptMessage()*. This might seem unnecessary but it keeps the code clean as designed and separates the classes from each other to make the development of new EncryptionManagers and Connectors easier.

```
auto& srcEvent = **timelineItem;
    bool isDecrypted = false;
    RoomEventPtr decrypted;
    if (auto e = eventCast<const EncryptedEvent>(&srcEvent))
    {
        decrypted = m_connector->decryptMessage(*e);
        if (decrypted) {
            decrypted->setSender(e->senderId());
            if (auto* msgEvent = eventCast<RoomMessageEvent>(decrypted)) {
                isDecrypted = true;
            }
        }
    }
const auto& event = isDecrypted ? *decrypted.get() : srcEvent;
```

Listing 2. The code that determines if the received Event is encrypted or just a regular plaintext one.

If the decryption was succesful the decrypted Event is casted in to a RoomMessageEvent, another subclass of Event, and it is handled normally after that. If the Event was not EncryptedEvent the original *srcEvent* variable is used and the Event is processed normally.

The EncryptionManager will then call the libQuotients *decryptMessage()* function which will start the decryption process coded in to the library. Since this was still a work in progress it obviously had some issues right off the bat.

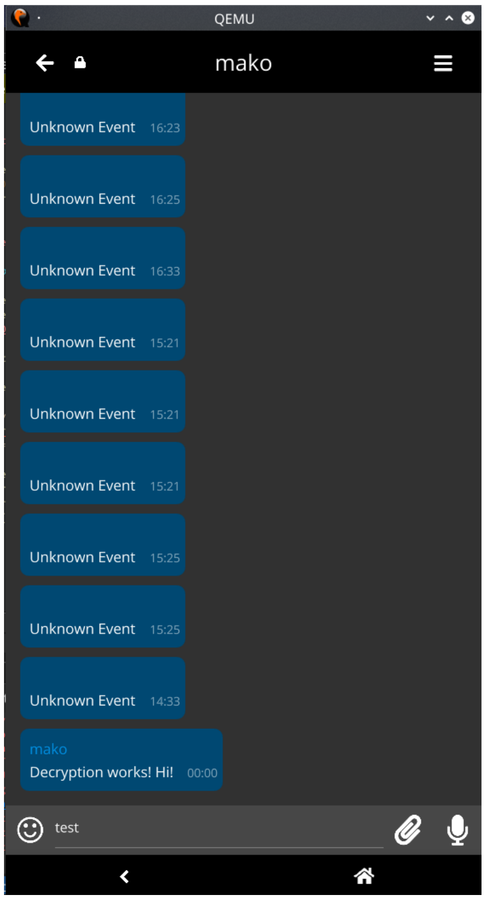
Matrix spec states that when an encrypted message is sent the process goes like this: First try to encrypt the message with Megolm, but if there is no active Megolm session, create a new one, then try to share Megolm session data with the receiving device encrypted with an Olm session, if there is no Olm session, initiate a new one, encrypt the Megolm session keys, send them as a to_device event and finally encrypt the message and send it as a regular encrypted event. The receiving end on the other hand needs to catch the to_device event, try and decrypt it with the Olm session initiated by the sending party, extract the keys from the to_device event, save the session data. Then catch the encrypted event with the actual message and decrypt it by using the just received session data.

Now libQuotient had most of this implemented, but very poorly. There were null pointers, some errors from QtOlm were not properly handled and the one-time keys were poorly handled. First the whole application crashed without an error message. After some heavy debugging the error was found to be a null pointer returned if there was an error creating a new Olm session initiated by the sending party. In Listing 3 *d→sessionDecryptMessage()* function is called and it returns a RoomEventPtr. In the next line the *decryptEvent* pointer is being called as a parameter. The problem was that *sessionDecryptMessage()* calls for QtOlm libraries Olm session initiation function and returns a null pointer if there is an error. After the program returns from the function the *decryptedEvent* should be checked if its a null pointer before it is used.

```
RoomEventPtr decryptedEvent = d-
>sessionDecryptMessage(*encryptedEvent.get());
// since we are waiting for the RoomKeyEvent:
event_ptr_tt<RoomKeyEvent> roomKeyEvent =
    makeEvent<RoomKeyEvent>(decryptedEvent->fullJson());
```

Listing 3. Part of the decryption process that resulted in an unexpected crash.

After the null pointer was fixed another issue was, why would the QtOlm return with an error. The debugger said the error was ”BAD.MESSAGE.KEY.ID”. That error was caused by bad one-time key handling. Matrix spec states that each device should have approximately 50 one-time keys uploaded to the server so others can use them to initiate Olm sessions. LibQuotient generates too many keys and only uploades some of them and discards the others. This results in following: the Riot.im client downloads an uploaded one-time key and uses it to create a new Olm session. Meanwhile the libQuotient based client has discarded the keys that were uploaded to the server so libQuotient cannot initiate the same session with the sending party because it thinks the keys are invalid. Once this was fixed by adjusting the key uploading process. The client can decrypt the messages sent encrypted from Riot.im (Figure 15).


QEMU

mako

Unknown Event 16:23

Unknown Event 16:25

Unknown Event 16:33

Unknown Event 15:21

Unknown Event 15:21

Unknown Event 15:21

Unknown Event 15:25

Unknown Event 15:25

Unknown Event 14:33

mako
Decryption works! Hi! 00:00

test

Figure 15: Screenshot of the decrypted message sent from Riot.im.

All the Unknown Events in the Figure 15 are events that haven’t been and cannot be decrypted, because the Megolm session data is not saved currently when the application is closed.

5.5.2 Encryption

For the encryption the IEncryptionManager interface was used with the intention that the encryption could be tested without messing with the libQuotient library and once encryption works properly, copy the changes to the library and make a pull request to the libQuotients git repository so their work is moved forward a little bit.

Because there is nothing done for encryption on libQuotient, it is required to use QtOlm directly. There is no documentation whatsoever for QtOlm so everything needs to be learned by reading the code. [31]

So the work starts by studying how to concretely handle the encryption process from Matrix spec point of view:

- First a Megolm session needs to be initiated if it doesn't exist by calling *olm_init_outbound_group_session* from the Olm library. This is done by creating a new OutboundGroupSession object in the QtOlm library.
- The Olm library will create the session id and session key. Get them from the OutboundGroupSession object.
- Download device list and their respective identity keys participating in the room and share the session data with every device as a *m.room_key* event using Olm session.
- Check if there already is an Olm session with a device.
- If not, start a new Olm session by claiming one of the devices one-time keys from the server with a HTTP request to /keys/claim API endpoint
- Create a new Olm session by calling *olm_create_outbound_session* from the Olm library. This is done by creating a new OutboundSession object from the QtOlm library and passing the keys for it as constructor parameters.
- Then build an encryption JSON payload as stated in the Matrix spec and encrypt it by calling *olm_encrypt* from the Olm library or just *encrypt()* in QtOlm OutboundSession object.
- Package that payload into an a *m.room.encrypted* event.
- Send that event to all of the devices participating in the room as a *to_device* event.
- Create an encryption payload and encrypt it using *olm_group_encrypt* or with QtOlm by calling the OutboundGroupSession *encrypt()* function.
- Finally wrap that encrypted payload into a *m.room.encrypted* event and send it to the server as regular event.

There are some prerequisites here that need to be addressed before it is possible to properly start tackling the checklist. First the application needs to keep track of users, their devices, device's identity and one-time keys. Users are easy as they are saved in libQuotient. Devices and identity keys however are not. They need to be queried from the server and libQuotient does not have ready made functions for it. They have classes mapping all the API endpoints of Matrix spec and generic API querying class so it is possible to use those to query the devices and identity keys. All of these need to be saved in the application. Ideally they would be saved in a database where they persist even when the application is closed but for now it is suitable to create a datastructure that is kept in the application memory. The datastructure ended up being a triple hash map, which is not very efficient but does the job for now. The hash map is as follows *QHash<QString, QHash<QString, QHash<QString, DeviceKeys*>>>** In detail its a pointer to a hash map of roomIds to userIds to deviceIds to a pointer into a custom struct DeviceKeys with the different keys that each device has.

Also, currently the application does not have any type of session handling so when the application is closed all the sessions are lost, but the sessions can be still managed while the app is open. Each room will have a Megolm session which can be used for a hash map to efficiently store roomId and Megolm session pairs and check if any given room has any sessions open. The same can be done for Olm sessions but each device has their own Olm session so the same mapping structure can be followed as above. In Listing 4 all the different data structures used can be seen.

```
QHash<QString, QHash<QString, QHash<QString, DeviceKeys*>>>*
m_roomIdToUsersToDevicesToKeys;
QHash<QString, QHash<QString, DeviceKeys*>> m_usersToDevicesToKeys;
QMap<QString, QtOlm::OutboundGroupSession*> m_megolmSessions;
QHash<QString, QHash<QString, QtOlm::OutboundSession*>> m_olmSessions;
```

Listing 4. All the necessary data structures to handle everything in application memory.

To populate these data structures the device list is needed and Matrix spec recommends to download the device list periodically. Other decent alternative is to download the devices list every time user enters to a room. Also, theres the option to download it once the user presses send but it is more efficient to do in the background when user switches rooms so the device list and keys are already downloaded before sending a message.

So, once the user switches to a room the function *downloadDevicesList()* is called (Listing 5). It first iterates over all the users in the room and gathers them to a QHash map and calls a libQuotients *callApi()* function which takes a Job class as a parameter. Jobs are special classes that map Matrix API endpoints to C++ classes so they are easier to manage in code. They are usually automatically generated and they transform the hash maps into JSON and they get sent as HTTP requests to the Matrix server.

```
/* Download devices and associated identity keys for each user
 */
void EncryptionManager::downloadDevicesList() {
    QHash<QString, QStringList> usersToDevices;
    for(auto user: m_conn->room(m_roomid)->users()) {
        usersToDevices.insert(user->id(), QStringList());
    }
    queryKeysJob = m_conn→callApi<Quotient::QueryKeysJob>(usersToDevices, 10000);
connect(queryKeysJob, &Quotient::BaseJob::success, this, [this] {
     QString deviceID;
     QHashIterator<QString, QHash<QString,
     Quotient::QueryKeysJob::DeviceInformation>> i(queryKeysJob->deviceKeys());
      // i.key() has the userid,
      // i.value() has another QHash with deviceId to all the info in the json of
the /keys/query endpoint
      while (i.hasNext()) {
          i.next();
           for(auto key : i.value().values()) {
                dID = key.deviceId;
                m_usersToDevices.insert(i.key(), i.value().keys());
                // Check if this device has been listed already
                DeviceKeys* deviceKeys = new DeviceKeys;
                deviceKeys->curve25519id = key.keys.value("curve25519:"+dID);
                deviceKeys->ed25519id = key.keys.value("ed25519:"+dID);
                deviceKeys->curve25519onetime = "";
                QHash<QString, DeviceKeys*> dk;
                dk.insert(deviceID, deviceKeys);
                m_usersToDevicesToKeys.insertMulti(i.key(), dk);
            }
        }
        m_roomIdToUsersToDevicesToKeys->insert(m_roomid, m_usersToDevicesToKeys);
        claimKeys();
    });
}
```

Listing 5. Function that downloads the devices list and associated keys and saves them to data structures.

Listing 5 also shows how to signalling system works in Qt. The function call *connect* is a special function in Qt which tells Qt what gets called when a signal gets triggered. In that specific call on the line 8 when *Quotient::BaseJob::success* signal fires it triggers the C++ lambda function with *this* as a context and *queryKeysJob* pointer as a parameter. It could also be done without the lambda function and instead make it call a regular function. Once the Job succeeds and a response is received, the response data is directly added to the *queryKeysJob* by the library and then the lambda function is executed.

In the lambda function all the necessary data is gathered in to the *m_roomIdToUsersToDevicesToKeys* hash map. Inside the lambda function a *claimKeys()* function is called to claim the one-time keys from the server and save them as well.

```
/* Claims the one-time keys from the server and saves them.
 */
void EncryptionManager::claimKeys() {
    QHash<QString, QHash<QString, QString>> usersToDevicesToAlgo;
    QHash<QString, QString> devicesToAlgo;
    QHashIterator<QString, QStringList> i(m_usersToDevices);
    // Check which devices dont have keys saved yet.
    while (i.hasNext()) {
        i.next();
        for(QString device : i.value()) {
            // Check if the keys have been claimed for the device

            devicesToAlgo.insert(device, "signed_curve25519");
            m_claimedDevices << device;
            qDebug() << "Claiming keys for: " << device;
        }
        usersToDevicesToAlgo.insert(i.key(), devicesToAlgo);
    }
    claimKeysJob = m_conn-
>callApi<Quotient::ClaimKeysJob>(usersToDevicesToAlgo, 10000);
    connect(claimKeysJob, &Quotient::BaseJob::success, this, [this] {
        QHashIterator<QString, QHash<QString, QVariant>> i(claimKeysJob-
>oneTimeKeys());
        // i.key() is the username
        // i.value() is a QHash with deviceId to one-time key and signature
        while (i.hasNext()) {
            i.next();
            QHashIterator<QString, QVariant> j(i.value());
            while(j.hasNext()) {
                j.next();
                QHash<QString, QVariant> map = j.value().toHash();
                for(auto key : map.values()) {
                    m_deviceOnetimeKey.insert(j.key(),
                                     key.toHash().value("key").toByteArray());
                }
            }
        }
    });
}
```

Listing 6. Function that claims one one-time key from the server for each device.

The *claimKeys()* function (Listing 6) is very similar to the *downloadDevicesList()* function. It this time creates a ClaimKeysJob and again connects a lambda function to the success signal. In the lambda function the one-time keys are gathered for creating new Olm sessions.

Now that the data structures are done, the first bullet point can be tackled. So, start by checking if there is a Megolm session and if none can be found create a new one. After which the session id and key can be extracted for later encryption (Listing 7).

```
// Check if there are existing megolm sessions
   if(!m_megolmSessions.contains(m_roomid)) {
       OutboundGroupSession* newMegolmSess;
       try {
           newMegolmSess = new OutboundGroupSession();
       } catch (OlmError* e) {
           qDebug() << "Error starting new outbound group session"
                    << e->what();
           return;
       }
       QString sessionId = newMegolmSess->id();
       QByteArray sessionKey = newMegolmSess→sessionKey();
```

Listing 7. Create a new Megolm session and get the session id and key that were generated.

Then by iterating through the users and devices map the devices which have an existing Olm session can be checked and new sessions can be created for those which do not have an active one. Also the identity and one-time keys can be gathered from the data structure since they are needed for creating new Olm sessions.

Once the Olm session is started, the *to_device* event can be crafted that first holds all the necessary data for the recipient to initialize Olm session and then the session data for the Megolm session used for future encryptions and decryptions of messages. Ultimately the *to_device* event JSON payload will look like Listing 8 except the

```
// m.room.encrypted event
{
  "type": "m.room.encrypted",
  "content": {
    "algorithm": "m.olm.v1.curve25519-aes-sha2",
    "sender_key": "<sender_curve25519_key>",
    "ciphertext": {
      "<device_curve25519_key>": {
        "type": 0,
        "body": {

          // m.room_key event starts
          "type": "m.room_key",
          "content": {
            "algorithm": "m.megolm.v1.aes-sha2",
            "room_id": "!Cuyf34gef24t:localhost",
            "session_id": "X3lUlvLELLYxeTx4yOVu6UDpasGEVO0Jbu+QFnm0cKQ",
            "session_key": "AgAAAADxKHa9uFxcXzwYoNueL5Xqi69IkD4sni8LlfJL7…"
          },
          "sender": "<sender_user_id>",
          "recipient": "<recipient_user_id>",
          "recipient_keys": {
            "ed25519": "<our_ed25519_key>"
          },
          "keys": {
            "ed25519": "<sender_ed25519_key>"
            }
          }
         // m.room_key event ends

      }
    }
  }
}
```

Listing 8. The *to_device* event JSON structure that contains the Megolm session data in plaintext

In Listing 8 the whole JSON structure is in plaintext but in reality the *m.room_key* event is encrypted with Olm. Because the encryption functions are not built into the libQuotient library and there is not yet a class that creates this type of event structure automatically, it is necessary to create a the JSON payload with Qt tools and wrap it in a generic Event and send it to the server. Listing 9 shows how the whole payload is constructed and sent to the server.

```
/ Construct the json payload for room_key event
    QJsonObject keys;
    keys.insert("algorithm", Quotient::MegolmV1AesSha2AlgoKey);
    keys.insert("room_id", m_roomid);
    keys.insert("session_id", QString(sessionId));
    keys.insert("session_key", QString(sessionKey));
    QJsonObject recipEd25519;
    recipEd25519.insert("ed25519", QString(ed25519id));
    QJsonObject ourEd25519;
    ourEd25519.insert("ed25519", QString(m_conn->olmAccount()-
>ed25519IdentityKey()));
    QJsonObject payload;
    payload.insert("content", keys);
    payload.insert("sender", m_conn->user()->id());
    payload.insert("recipient", recipient);
    payload.insert("recipient_keys", recipEd25519);
    payload.insert("keys", ourEd25519);
    payload.insert("type", "m.room_key");
    QJsonDocument json(payload);
    // Encrypt this payload with olm
    QByteArray cipherText;
    try {
        cipherText = olmSession->encrypt(json.toJson())->cipherText();
    } catch (OlmError* e) {
        qDebug() << "Error encrypting payload"
                 << e->what();
        return;
    }
    // Construct the encrypted event with encrypted room_key event inside
    QJsonObject keyJson;
    keyJson.insert("type", 0);
    keyJson.insert("body", QString(cipherText));
    QJsonObject ciphertextJson;
    ciphertextJson.insert(curve25519id, keyJson);
    QJsonObject finalPayload;
    finalPayload.insert("algorithm", "m.olm.v1.curve25519-aes-sha2");
    finalPayload.insert("sender_key", QString(m_conn->olmAccount()-
>curve25519IdentityKey()));
    finalPayload.insert("ciphertext", ciphertextJson);
    const Quotient::Event* evt = new
Quotient::Event(Quotient::typeId<void>(),"m.room_key", finalPayload);
    std::unordered_map<QString, std::unordered_map<QString, const
Quotient::Event*>> usersToDevicesToEvents;
    std::unordered_map<QString, const Quotient::Event*> deviceEvent;
    deviceEvent[recipDeviceId] = evt;
    usersToDevicesToEvents[recipient] = deviceEvent;
    m_conn->sendToDevices("m.room.encrypted", usersToDevicesToEvents);
```

Listing 9.	Generation of JSON payload for the *to_device* event.

After the *to_device* event is finally succesfully sent, the actual encrypted Megolm message can be generated, encrypted and sent. This is done by calling *doEncryptMessage()* function (Listing 10) in the EncryptionManager to do the actual encrypting of the message. Everything up to this point has been just preparation to set up the sessions.

```
void EncryptionManager::doEncryptMessage(QString plaintext, QString sessionId,
QtOlm::OutboundGroupSession* megolmSession) {
    using namespace QtOlm;
    QByteArray cipherText2;
    Quotient::RoomMessageEvent msgEvt(plaintext,
Quotient::MessageEventType::Text);
    msgEvt.setRoomId(m_roomid);
    try {
        cipherText2 = megolmSession->encrypt(msgEvt.originalJson());
    } catch (OlmError* e) {
        qDebug() << "Error encrypting payload"
                 << e->what();
        return;
    }
    Quotient::EncryptedEvent* encEvt = new
Quotient::EncryptedEvent(cipherText2,
                          m_conn->olmAccount()->curve25519IdentityKey(),
                          m_conn->deviceId(), sessionId);
    m_conn→room(m_roomid)→postEvent(encEvt);
}
```

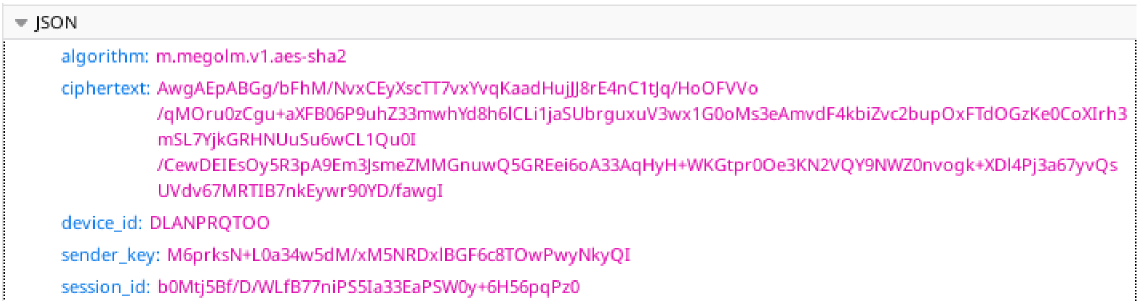
Listing 10. The *doEncryptMessage()* function that does the actual message encryption.

In this function a RoomMessageEvent class is used to do the JSON generation then the JSON is encrypted using the established Megolm session. Then again an EncryptedEvent class is used to create all the necessary JSON for the request and the event is sent to the room.

Now that all the setting up is done and the sessions have been created no new sessions are created unless the user start conversing with a new user or device.

5.6 Verification of Results

It is necessary to verify the results of the encryption to be sure that no data leaks in the conversation. The decryption is fairly simple to verify. With browsers developer tools the encrypted payload that Riot.im sends can be seen and in the application the ciphertext can be seen as a readable plaintext. Figure 16 shows what the developer tools caught.

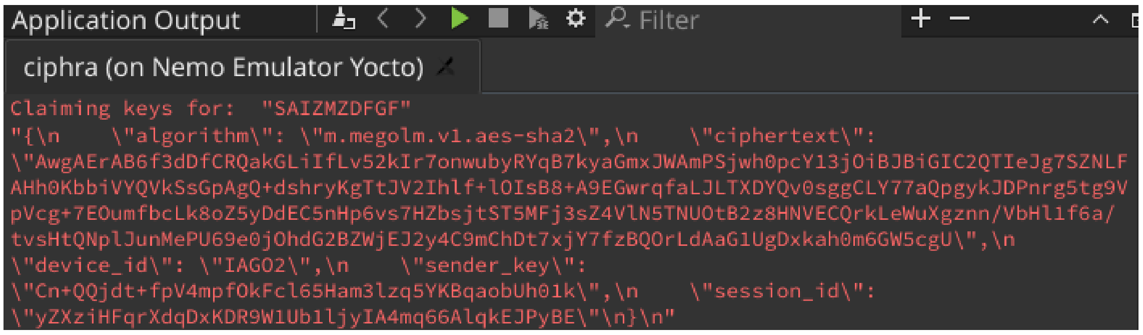


```
JSON
    algorithm: m.megolm.v1.aes-sha2
    ciphertext: AwgAEpABGg/bFhM/NvxCEyXscTT7vxYvqKaadHujJJ8rE4nC1tJq/HoOFVVo
                /qMOru0zCgu+aXFB06P9uhZ33mwhYd8h6lCLi1jaSUbrguxuV3wx1G0oMs3eAmvdF4kbiZvc2bupOxFTdOGzKe0CoXIrh3
                mSL7YjkGRHNUuSu6wCL1Qu0I
                /CewDEIEsOy5R3pA9Em3JsmeZMMGnuwQ5GREei6oA33AqHyH+WKGtpr0Oe3KN2VQY9NWZ0nvogk+XDl4Pj3a67yvQs
                UVdv67MRTIB7nkEywr90YD/fawgI
    device_id: DLANPRQTOO
    sender_key: M6prksN+L0a34w5dM/xM5NRDxlBGF6c8TOwPwyNkyQI
    session_id: b0Mtj5Bf/D/WLfB77niPS5Ia33EaPSW0y+6H56pqPz0
```

Figure 16: JSON payload sent from Riot.im.

The ciphertext is clearly a garbled mess and the application decrypts it perfectly. Everything seems to be as its supposed to be for the decryption.

For inspecting the encryption from the QEMU virtual emulator is a bit tougher but there are tools to do it. One option could be to take a *tcpdump* on the emulator and save it somewhere and afterwards inspect it with a packet inspection tool like WireShark. Unfortunately the emulator used here did not come with a *tcpdump*. Easiest way to inspect the payload for this specific case is to print it out in the debugging console (Figure 17).



```
Claiming keys for:  "SAIZMZDFGF"
"{\n    \"algorithm\": \"m.megolm.v1.aes-sha2\",\n    \"ciphertext\":
\"AwgAErAB6f3dDfCRQakGLiIfLv52kIr7onwubyRYqB7kyaGmxJWAmPSjwh0pcY13jOiBJBiGIC2QTIeJg7SZNLF
AHh0KbbiVYQVkSsGpAgQ+dshryKgTtJV2Ihlf+lOIsB8+A9EGwrqfaLJLTXDYQv0sggCLY77aQpgykJDPnrg5tg9V
pVcg+7EOumfbcLk8oZ5yDdEC5nHp6vs7HZbsjtST5MFj3sZ4VlN5TNUOtB2z8HNVECQrkLeWuXgznn/VbHl1f6a/
tvsHtQNplJunMePU69e0jOhdG2BZWjEJ2y4C9mChDt7xjY7fzBQOrLdAaG1UgDxkah0m6GW5cgU\",\n
\"device_id\": \"IAGO2\",\n    \"sender_key\":
\"Cn+QQjdt+fpV4mpfOkFcl65Ham3lzq5YKBqaobUh01k\",\n    \"session_id\":
\"yZXziHFqrXdqDxKDR9W1Ub1ljyIA4mq66AlqkEJPyBE\"\n}\n"
```

Figure 17: JSON payload sent from the application.

Figure 17 shows that the JSON payload was similarly encrypted as Riot.im did. Ciphertext is similar garbled mess.

6 Results

In this thesis the goal was to take a deeper look into cryptography from mathematical point of view, understand the architecture of Matrix Open Standard and develop deeper understanding for modern encryption algorithms used in messaging applications by concretely implementing one. The concrete end goal was to build upon a messaging application that had the basic messaging functions and add encryption to it in a way that would allow for it to be changed for another encryption scheme without too much work on the core messaging.

In the end the mathematical theory of cryptography was a very interesting to delve into, but it was way bigger of a field than expected. The rabbit hole just never ended with new and interesting topics emerging every time the subject was delved deeper into. For that reason this thesis could not possibly tackle every interesting topic of cryptography. Navigating the subject and picking out the necessary topics for the goal of the thesis proved difficult without simplifying or flat out leaving out things. For example key generation and key shceduling was not touched in the encryption schemes that would require deeper knowledge of random number generation and Pseudo-Random Number Generators (PRNG).

As always with coding projects the concrete work proved to be more difficult compared to what was imagined before starting the work. Even though Qt was familiar before starting the thesis, the C++ code was way more complex compared to what was experienced before. Also the no documentation for the libraries part required building understanding for the libraries through reading the code. This was a blessing and curse. While it perhaps gave deeper understanding of the code and taught the value of understanding code, it also was time consuming and difficult at times. The coding issues in the libraries also required to delve into the library itself and search and fix bugs inside it. However this resulted in being in contact with the author and he was happy to know the issues currently in the library. He also encouraged to contribute by creating pull requests for the library.

The implementation of encryption was exactly what was asked, but there are definitely multiple improvements that weren’t able to be addressed given the time frame. Currently the implementation is great for high security situations where everything is handled in memory and nothing is saved persistently on the disk which stops certain security risks. This however can also be an usability issue where all the encrypted messages are lost once the application is closed. There were also bugs in the libraries key handling which weren’t completely fixed since they were sligthly out of scope for this project.

Overall the study was great and interesting. Most if not all the goals were met given the timeframe. Deep understanding of cryptography was developed which will come in handy in the future. The code design with interfaced implementation was appreciated and the company was happy with the result and it will be used as a great base for future developments. The study of Matrix Open Standard and the modern encryption algorithms were greatly appreciated and gave excellent insight for the company to develop their products further.

References

1 Pandya, Dwiti; Khushboo Ram Narayan; Sneha Thakkar; Tanvi Madhekar, B.S Thakare. 2015. Brief History of Encryption. Online. <https://www.researchgate.net/publication/289124401_Brief_History_of_Encryption> Accessed 11.09.2019

2 Crypto Museum. One-Time Pad, The unbreakable code. Online. <https://www.cryptomuseum.com/crypto/otp/index.htm> Accessed 16.09.2019

3 Katz, Jonathan & Lindell, Yehuda. 2007. Introduction to Modern Cryptography.

4 Dworkin, Morris. 2001. Recommendations for Block Cipher Modes of Operation. NIST National Institute of Standards and Technology. Online <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf> Accessed 18.09.2019

5 Data Encryption Standard (DES). Online. <https://academic.csuohio.edu/yuc/security/Chapter_06_Data_Encription_Standard.pdf> Accessed 19.09.2019

6 Burr, William. NIST National Institure of Standards and Technology. Online. <https://nvlpubs.nist.gov/nistpubs/sp958-lide/250-253.pdf> Accessed 19.09.2019

7 Chu, Edward; Kim, Paul; Kim, Phillip; Liu, Frank; Sharma, Jason; Yu Jeffrey. 2000. The Selection of the Encryption Standard. MIT. Online <http://web.mit.edu/6.933/www/Fall2000/aes/AES_final_6933.pdf> Accessed 20.09.2019

8 Pathan, Al-Sakib Khan & Azad, Saiful. 2014. Practical Cryptography. E-book. O'REILLY.

9 Diffie, Whitfield; Hellman, Martin E. 1976. New Directions in Cryptography. Insitute of Electricar and Electronics Engineering. Online. <https://ee.stanford.edu/~hellman/publications/24.pdf> Accessed 09.10.2019

10 Schneier, Bruce & Ferguson Niels. 2003. Practical Cryptography.

11 Kallam, Sivanagaswathi. 2015. Diffie-Helmann: Key Exchange and Public Key Cryptosystems. Indiana State University. Online. <http://cs.indstate.edu/~skallam/doc.pdf> Accessed 09.10.2019

12 Matrix. Frequently Asked Question. Online. <https://matrix.org/faq/> Accessed 14.10.2019

13 Matrix. Who is Matrix.org. Archived. Online. <https://web.archive.org/web/20190329152401/https://matrix.org/blog/who-is-matrix-org/> Accessed 14.10.2019

14 Amdocs. Amdocs Unified Communications Solution. Archived. Online. <https://web.archive.org/web/20141003202858/http://www.amdocs.com:80/Products/digital-lifestyle-services/Pages/unified-communications.aspx> Accessed 14.10.2019

15 Rogers, Steward. 2018. Status invests $5 million in Matrix to create a blockchain messaging superpower. VentureBeat. Online. <https://venturebeat.com/2018/01/29/status-invests-5-million-in-matrix-to-create-a-blockchain-messaging-superpower/> Accessed 14.10.2019

16 Hodgson, Matthew. 2018. Matrix and Riot confirmed as the basis for France's Secure Instant Messenger app. Online. <https://matrix.org/blog/2018/04/26/matrix-and-riot-confirmed-as-the-basis-for-frances-secure-instant-messenger-app> Accessed 14.10.2019

17 Rosemain, Mathieu. 2018. France builds WhatsApp rival due to surveillance risk. Reuters. Online. <https://www.reuters.com/article/us-france-privacy/france-builds-whatsapp-rival-due-to-surveillance-risk-idUSKBN1HN258> Accessed 14.10.2019

18 Lomas, Natasha. 2019. New Vector scores $8.5M to plug more users into its open, decentralized messaging Matrix. Techcrunch. <https://techcrunch.com/2019/10/10/new-vector-scores-8-5m-to-plug-more-users-into-its-open-decentralized-messaging-matrix/> Accessed 14.10.2019

19 Hodgson, Matthew. 2019. New Vector raises $8.5M to accelerate Matrix/Riot/Modular. Online. <https://matrix.org/blog/2019/10/10/new-vector-raises-8-5-m-to-accelerate-matrix-riot-modular> Accessed 14.10.2019

20 Matrix. Matrix Specification. Online. <https://matrix.org/docs/spec/> Accessed 14.10.2019

21 W3. Relationship to the World Wide Web and REST Architectures. Online. <https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest> Accessed 15.10.2019

22 Matrix Guide. Application Services. <https://matrix.org/docs/guides/application-services> Accessed 15.10.2019

23 Marlinspike, Moxie; Perrin Trevor (editor). 2016. The Double Ratchet Algorithm. Revision 1. Online. <https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf> Accessed 13.11.2019

24 Matrix. Olm: A Cryptographic Ratchet. Online. <https://gitlab.matrix.org/matrix-org/olm/blob/master/docs/olm.md> Accessed 13.11.2019

25 Matrix. Megolm group ratchet. Online. <https://gitlab.matrix.org/matrix-org/olm/blob/master/docs/megolm.md> Accessed 13.11.2019

26 Qt. About Qt. Online. <https://wiki.qt.io/About_Qt> Accessed 13.11.2019

27 Qt. Nokia-and-Qt. Online. <https://wiki.qt.io/Nokia-and-Qt> Accessed 13.11.2019

28 Levine, Yasha. 2018. The Surveillance Valley.

29 Matrix. End-to-End Encryption implementation guide. <https://matrix.org/docs/guides/end-to-end-encryption-implementation-guide/> Accessed 14.11.2019

30 Quotient-IM. libQuotient library. Online. <https://github.com/quotient-im/libQuotient> Accessed 14.11.2019

31 QtOlm. libQtOlm library. Online. <https://gitlab.com/b0/libqtolm> Accessed 14.11.2019

32 Wikimedia. Cipher Block Chaining mode. <https://upload.wikimedia.org/wikipedia/commons/8/80/CBC_encryption.svg?download> Accessed 15.11.2019

33 Wikimedia. Feistel Cipher Diagram. <https://upload.wikimedia.org/wikipedia/commons/f/fa/Feistel_cipher_diagram_en.svg?download> Accessed 15.11.2019

34 Hodgson, Matthew. 2015. Matrix - One-year in. Slides. Online. <https://www.slideshare.net/jaquayle/matrix-oneyear-in-matthew-hodgson-matrixorg> Accessed 15.11.2019

35 Qt. Signals & Slots. Online <https://doc.qt.io/qt-5/signalsandslots.html> Accessed 20.11.2019

36 Schneier, Bruce. 2009. Blog. Online. <https://www.schneier.com/blog/archives/2009/07/another_new_aes.html> Accessed 20.11.2019

37 Bernstein, Daniel J. 2007. The Salsa20 family of stream ciphers. Online. <http://cr.yp.to/snuffle/salsafamily-20071225.pdf> Accessed 20.11.2019