ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

Blockchain-based end-to-end encryption for Matrix instant messaging

Relatore:
Chiar.mo Prof.
Stefano Ferretti

Correlatore:
Chiar.mo
Mirko Zichichi

Presentata da:
Julian Sparber

I Sessione: 15/07/2020
Anno Accademico 2019/2020

I dedicate this work to everyone who helped me in the past or will help me in the future to write a thesis, a blogpost, an email or any written text. Thanks to them I was able to overcome the challenges that my dyslexia posed and get this far in my academic career.

Abstract

Privacy and security in online communication is an important topic today, especially in the context of instant messaging. A lot of progress has been made in recent years to ensure that conversations are secure against attacks by third parties, but privacy from the service provider itself remains difficult. There are a number of solutions offering end-to-end encryption, but most of them rely on a centralized server, proprietary clients, or both.

In order to have fully secure instant messaging conversations, a decentralized and end-to-end encrypted communication protocol is needed. This means there is no single point of control, and each message is encryped directly on the user's device such that only the recipient can decrypt it.

This work proposes an end-to-end encryption system for the Matrix protocol based on blockchain technology. Matrix is a decentralized protocol and network for real-time communication that is currently mostly used for instant messaging. This protocol was selected because of its versatility and extensibility.

Using the Secret Store feature in OpenEthereum, the proposed system encrypts data using keys stored on the Ethereum blockchain. Access control to the keys is also handled by the Secret Store via a smart contract. The proposed encryption system has multiple advantages over alternative schemes: The underlying blockchain technology reduces the risk of data loss because of its decentralized and distributed nature. Thanks to the use of smart contracts this system also allows for the creation of an advanced access control system to decryption keys.

In order to test and analyze the proposed design, a reference implementation was created in the form of a library. This library can be used for future research, but also as a building block for different applications to easily implement end-to-end encryption based on blockchain technology.

Sommario

Privacy e sicurezza sono tematiche sempre più importanti nelle comunicazioni online. Questa tesi si concentra sulla sicurezza della messaggistica istantanea. Un importante strumento per sostenere una conversazione digitale confidenziale è la cifratura end-to-end, che permette di cifrare un messaggio direttamente sul dispositivo dell'utente e di decriptarlo solo sul dispositivo del destinatario. In questo modo se ne nasconde il contenuto a ogni altra entità, incluso il fornitore di servizi Internet e l'azienda che offre il servizio di comunicazione.
In questo lavoro è proposto un nuovo sistema per la cifratura end-to-end utilizzando Ethereum, una piattaforma basata su blockchain per eseguire, in maniera distribuita, operazioni computazionali. Più nello specifico Ethereum è una tecnologia che permette di eseguire codice attraverso gli smart contract. Il sistema proposto di cifratura end-to-end usa il Secret Store, una funzionalità integrata nel client Ethereum chiamato OpenEthereum(Parity). Questo strumento permette di generare le chiavi crittografiche necessarie per il funzionamento del sistema, in più le immagazzina in maniera decentralizzata e distribuita e ne regola l'accesso attraverso uno smart contract.
Matrix è un sistema per la comunicazione in tempo reale, oggi principalmente usato per la messaggistica istantanea. È un sistema decentralizzato, creato appositamente per essere esteso con nuove funzionalità e pertanto è stato scelto come protocollo di trasporto per questo progetto. Matrix ha già un sistema di cifratura end-to-end, ma il nuovo sistema proposto ha un numero significativo di vantaggi.

Uno di questi vantaggi è la separazione tra il processo di cifratura e il trasporto del messaggio, che permette di riutilizzare lo stesso processo di cifratura per un protocollo diverso da Matrix. Inoltre, grazie all'astrazione del sistema, sarebbe facile integrare i frutti di questo lavoro in un qualsiasi software per implementare la funzionalità di cifratura end-to-end.
Ethereum, Secret Store e Matrix sono tutti sistemi decentralizzati e per questo nessuna parte del sistema proposto deve fare affidamento a una singola entità per svolgere le varie funzionalità richieste. Il sistema proposto non ha necessità di backup grazie alla tecnologia blockchain e finché l'utente mantiene i permessi d'accesso corretti, impostati via smart contract, non esiste alcuna possibilità di perdere l'accesso ai messaggi inviati. In più, sempre tramite gli smart contract, è possibile creare un sistema avanzato per il controllo di accesso alle chiavi di decriptazione, che può essere utilizzato per aggiungere nuovi utenti a una conversazione oppure per ritirare l'accesso al suo contenuto in un qualsiasi momento.
In questa tesi è proposto un sistema funzionante di cifratura end-to-end basato sulle garanzie di sicurezza di Ethereum e di blockchain, per testare il quale è stata scritta una libreria specifica che potrebbe essere riutilizzata, grazie al suo design, per altri protocolli di comunicazione. In conclusione, questa tesi stabilisce una base per la futura ricerca su sistemi di cifratura end-to-end basati su blockchain.

Introduction

Digital communication is becoming ever more important in today's modern society. We are increasingly relying on it for many critical parts of our everyday lives. Since digital communication is now a basic need, we need ways to secure and protect the interaction and the system itself. It's not enough to encrypt its content, but we also need resilience against system failures and attacks. Furthermore, it's important not to only rely on the trustworthiness of a third party to guarantee the integrity and the privacy of the system. A good start is to rely as much as possible on open source software, so that it can be audited and improved by anyone with the skills to do so.

This work focuses mainly on instant messaging, or IM for short. The most basic feature of an instant messenger is real time text communication, but nowadays most people expect much more advanced features such as image and file sharing, voice messages, and video chat.

Over the past decade the public's concern about their online privacy has grown significantly, and as a result people want better privacy and security online. Though security has generally improved as a result, not enough progress has been made to better protect user's privacy.

Most of the messaging systems used every day by the general public are centralized services, with one single point of control and hence also a single point of failure. The most popular services all require the user to trust their system, without any real possibilities to review or validate what the provider is actually doing with their data. Most services offer some kind of encryption

to protect user data, but only a few provide encryption which inhibits access to the conversation content by the provider themselves. These systems may protect the user against outside eavesdropping or other types of attacks, but not so much against eavesdropping from inside the system. Therefore, the user must trust the service provider blindly.
The basic encryption protocols many services have put in place are generally built on top of solid cryptographic foundations that considerably improve the status quo, but often their systems are proprietary and thus the user is still required to fully trust the company and hope that the encryption was implemented competently. So far, most companies addressed only the basic protection, but have yet to mitigate the privacy issue. As a result, users are still left with no power and not much protection.
This is where Matrix comes into play. Matrix is a decentralized communication protocol, designed to not require any centralized server. Therefore it does not have a single point of failure and, more importantly, a single point of control. Its main use case today is instant messaging, but it was created to accommodate any type of real-time communication[1]. The entire system is open-source, which means that its source code can be reviewed by anybody and it is developed by a globally distributed community, composed of many different companies and individuals. Matrix also has comprehensive public documentation, which helps the development and allows users to understand and work on the system.
Any user can run their own server, or join an existing one, through which they can join the Matrix network. It is possible to run a completely separate network, which can join the main network at a later point, or stay isolated forever. Obviously not everybody has the technical know-how or interest to set up and run their own server. There are already many existing servers, many of them free of charge, on which a user can create an account and participate in communications on the Matrix network. This adds up to the same problem as described previously, but it's slightly better because the user's data is stored only on their server or the server the people they com-

municate with are on. Still, a server involved in a conversation can read the messages sent between different users. The only way to circumvent this issue is end-to-end encryption. This means that a user encrypts their messages on their own device and only the recipients, selected by the sender, can decrypt them and read the messages content. Matrix implements an end-to-end encryption system to encrypt the entire communication[2].
The scope of this work is to create an alternative end-to-end encryption system which replaces the default end-to-end encryption system used by Matrix. The proposed system is built on top of the blockchain-based distributed computing platform Ethereum, specifically on OpenEthereums's Secret Store. OpenEthereum (previously known as Parity) is an advanced Ethereum client, that implements a feature called Secret Store[3]. The Secret Store is a core technology which allows to generating keys for encryption and decryption, it provides distributed key storage, and has a key retrieval system which uses smart contracts for permission control[4].
The new end-to-end encryption system proposed in this work has the advantage that it fully separates the encryption of a message from its transport, because a message is encrypted without any dependency on the protocol that is used to send it. The Secret Store was selected, among other things, because it provides an abstraction over the underlying cryptographic primitives. In addition, it permits the creation of an advanced control system to access the encryption keys, and hence the conversation. This is especially interesting when a user can revoke the access to the content of a conversation. Thanks to the blockchain all the cryptographic keys are stored securely over a distributed network of nodes, which protects them from attackers and data loss. As long as the user maintains the correct access permissions via a smart contract loss of the message content is highly unlikely.
This document describes a functioning end-to-end encryption system that can be used to secure the communication among different parties. In order to enable this, a library was created which can be used as a foundation to add end-to-end encryption not only to the Matrix protocol, but also to

other transport protocols. The library is published under the GNU General Public License (GPL-3.0-or-later) and the source code can be found at https://github.com/jsparber/e2e-secretstore.

The first section of this document is dedicated to the explanation of fundamental concepts and to give background knowledge. The second part contains the proposed design, and a description of the reference implementation and evaluation of the system. The implementation was written to experiment, text and analyze the proposed design. This thesis is structured in the following chapters:

- Cryptography. This section gives a general overview about Cryptography and its importance. It explains the most important cryptographic schemes related to this work. All systems reported in this chapter are the foundation of the entire project, and they allow it to function as intended.
- Blockchain. This part gives some background on blockchain technology, including underlying principles, and the data structures it is buildt upon. It also explains how the technology is used in cryptocurrencies, such as Bitcoin and Ethereum.
- Ethereum. This chapter will talk about Ethereum more specifically. Ethereum is not just a cryptocurrency based on blockchain technology, but rather a distributed computing platform. This part also explains smart contracts, the core Ethereum feature used in this work.
- Matrix. A distributed real-time communication protocol. This part gives an overview of Matrix, describing the overall structure of the protocol, as well as core features and advantages. For completeness, the standard encryption method used in Matrix is also briefly explained.
- Proposed designs. This chapter describes the proposed design for the end-to-end encryption based on OpenEthereum's Secret Store for Ma-

trix. This section also describes problems of the system and discusses possible solutions.

- Reference implementation. To test the proposed design this work includes a reference implementation. This chapter talks about the tools and software used for the implementation and gives background knowledge, not included in other parts of this document. It explains the overall structure and outlines the most important decisions taken during the creation of the software.
- Analysis and evaluation. This section contains experiments to test the performance of the created end-to-end encryption system.

Contents

List of Figures

List of Tables

Chapter 1

Cryptography

In this chapter a general introduction to cryptography is given and it describes important cryptography schemes and primitives related to the proposed encryption system.
Historically cryptography, often referred to as classical cryptography, was considered to be an art form that tries to creatively create procedures and processes to transform understandable information into a gibberish message, so that no one who doesn't know the algorithm used to transform the information can still understand it. Before the 1980s, cryptography was essentially the process of creating algorithms, that satisfy the objective, mostly in an innovative and inventive manner[5]. Cryptography wasn't based on any scientific foundation and had no general definition to define what could be considered as a good and secure system. Today's cryptography on the other hand is built on solid mathematical constructs and it defines specific criteria to analyze the system and to determine its security.
Classical cryptography was only about encrypting information, the process of making a data unreadable for everyone except for the party who is desired to read it, intuitively, the receiver requires some information on how to obtain the original content. Encrypting is the process of translating a so called cleartext, the understandable message, into an unreadable message called ciphertext. The operation of converting the ciphertext back from its

unreadable state into a readable message is called decryption.
Back then, cryptography was used only by the military and governments[5]. But the modern systems, created by today's cryptographers, are used by millions of people every day, and it's the foundation of every secure digital activity.

Cryptography doesn't study anymore only how to encrypt and decrypt information, but it includes a much wider area of interest. Applications include identity verification, data integrity, digital currencies and much more. A core principle of modern cryptography on which all cryptographic schemes should be built on is:

Kerckhoffs's principle. *The cipher method must not be required to be secret, and it must be able to fall into the hands of the enemy without inconvenience[5].*

This principle says that an encryption scheme must be public and that security by obscurity is bad practice to provide a secure system. The secrecy guarantee needs to rely only on the secrecy of a key but not on the secrecy of the protocol itself.

One of the best and oldest known encryption schemes is the Caesars cipher. The scheme takes a sentence and rotates each character of the input string by a certain number of positions in the alphabet. Caesar always rotated by three positions, but to respect Kerckhoffs's principle let's assume that the key is variable.
Let's take the sentence "This is a top-secret sentence" as an example. When rotating each character by three positions in the alphabet the ciphertext "Xlmw mw e xst wigvix wirxirgi" is obtained. The resulting string looks pretty random at first glance and it's not possible to read the message. To return to the cleartext it's just the simple task of rotating each character back by three positions in the alphabet. Although the ciphertext looks like gibberish it actually gives quite some information about the original text. To analyze the security of the cipher, it's necessary to first define what the cipher

tries to accomplice and what security means in this context of this algorithm. Informally, the objective is that a ciphertext obtained by encrypting a message via Caesars cipher can't be decoded by anybody who doesn't know how the cipher works. A scheme is secure in this context if it satisfies the previous stated objective. It's relatively easy to see that Caesar's cipher doesn't

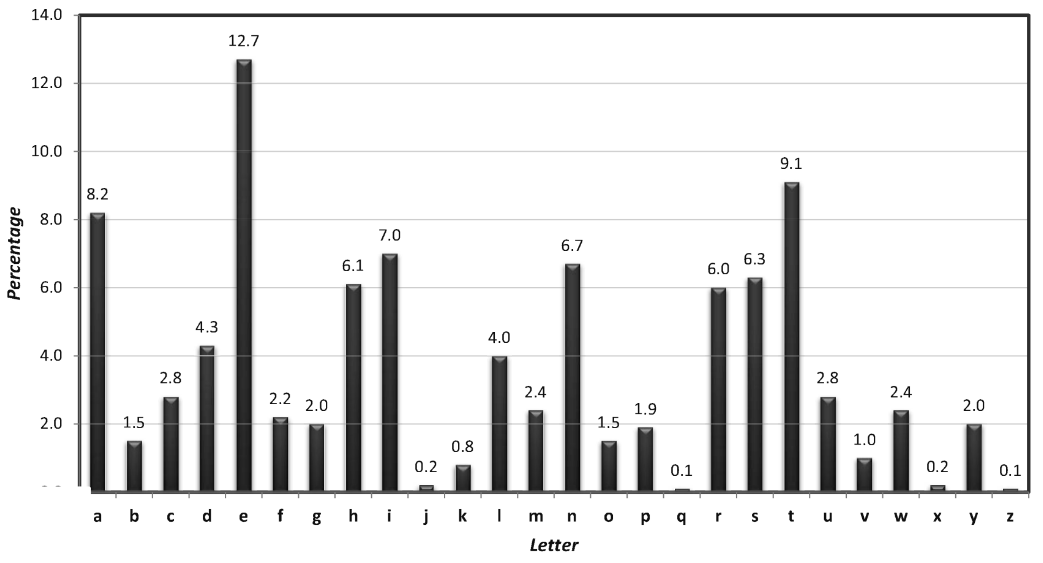


Figure 1.1: The frequency of each letter in the English language[5]

satisfy the objective. The scheme is relatively easy to break via statistical analysis attack. It's possible to use knowledge about the English language to easily find the key. The figure 1.1 shows the frequency of each letter in the English alphabet. Since the letter "e" is the most frequent letter in the English alphabet it's possible to link it to the most frequently used letter in the ciphertext, thus obtain the used key. Even without any knowledge about the frequency of the letters, it would be simple to gain access to the key by trying all 26 possible keys. This type of attack is called brute forcing. In this case a brute forcing attack can be done in less than a second via a computer, but also doing it by hand would be relatively quick.

1.1 Symmetric-key cryptography

Symmetric-key encryption, or sometimes called private-key encryption or also just encryption, is used to encrypt data via an encryption scheme that allows data to be encrypted and decrypted generally with the same key. It is often used to protect an information that is later retrieved by the same person for example disk encryption where it isn't required to share a key with anybody. If the data is intended to be shared with somebody else the key needs to be shared over a secure channel. Figure 1.2 shows a typical setup of a communication between two parties, Alice and Bob. Both have the same key in memory and hence they can access the message. An attacker, Eve, who is eavesdropping on the conversation between Alice and Bob can't read the message because they don't have access to the key used by Alice to encrypt the message.

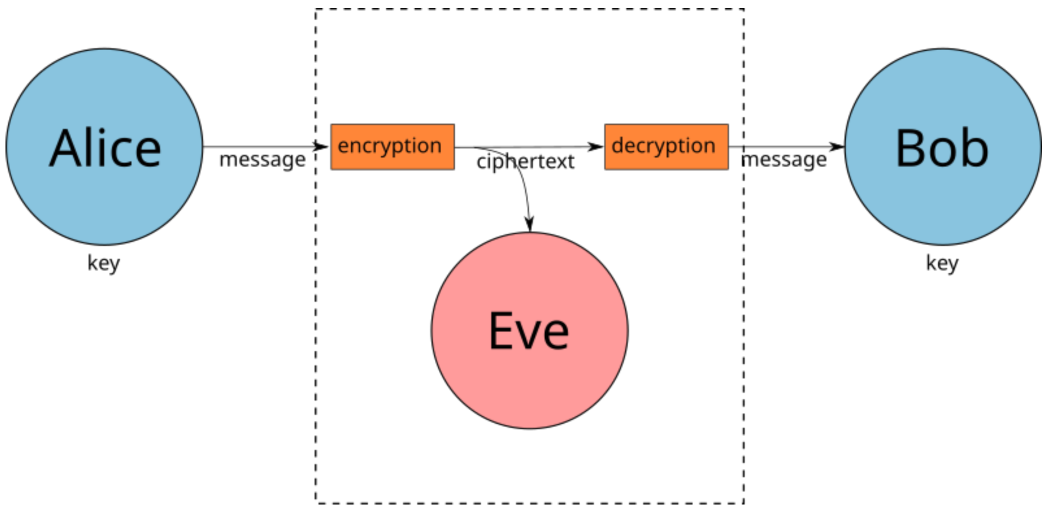


Figure 1.2: Private encryption scheme: Alice sends an encrypted message to Bob. Both parties use the same encryption key. Eve is some third party who only sees the ciphertext, they know the algorithm used for encryption and decryption but do not have access to the key and hence not to the message.

Symmetric ciphers can be divided into two types[6]:

- Stream cipher. This type of cipher encrypts each input bit individually by adding a bit from a key stream to the plaintext bit[6]. An example for this type of cipher is the Vigenère Cipher or also called substitution cipher, which works on the same principles as the Caesar's cipher, substituting letters but in a more secure way.

- Block cipher. Block ciphers on the other hand take a chunk, a block of the plaintext bits, and encrypted it with the key. This implies that the encryption of any plaintext bit depends on every other plaintext bit in the same block[6]. Advanced encryption standard (AES) uses for example a block length of 128 bits.

1.2 Public-key cryptography

Public-key cryptography is quite different from symmetric-key cryptography explained in the previous section. It's based on mathematical primitives which allow to generate two different keys. The first key, called public-key, is used to encrypt and the second key is used to decrypt data or a message. The public-key can be published without any restriction, also to an attacker. A widespread knowledge of the public-key can improve the overall security. This implies that no secure channel is needed to distribute any shared-secret, because there is none. The private-key is the secret part of the system and must be kept secure and shouldn't be shared with anybody.
Public-key ciphers are often used directly to allow different parties to communicate securely, but they can also be used to establish a secure communication channel so that a secret key can be shared, which has the advantage that it enables the possibility to use symmetric-key encryption that operate generally faster than this type of encryption.
Public-key cryptography isn't limited just to encrypting data, it is a primary ingredient to assure confidentiality, authenticity and non-repudiation in modern cryptography. It is used for many applications including identity validation, document signing and user authentication.

Figure 1.3 shows an example of a secure message exchange between Alice and Bob. Alice and Bob exchange their public key via some key exchange protocol, obviously they don't communicate the private part. Then Alice encrypts a message for Bob using Bob's public-key and sends it to them. Bob decrypts the obtained message via their private-key. An attacker Eve eavesdrops on the communication channel and obtains the public-key of Alice and Bob as well as the ciphertext. Eve can't decrypt the sent message because they would need the private-key it was encrypted for, but the attacker could encrypt their own message for Bob using the public-key they know. This system makes the content completely unreadable to an adversary, but it doesn't guarantee the authenticity of the sender. Public-key cryptography can be used to solve this problem as well.

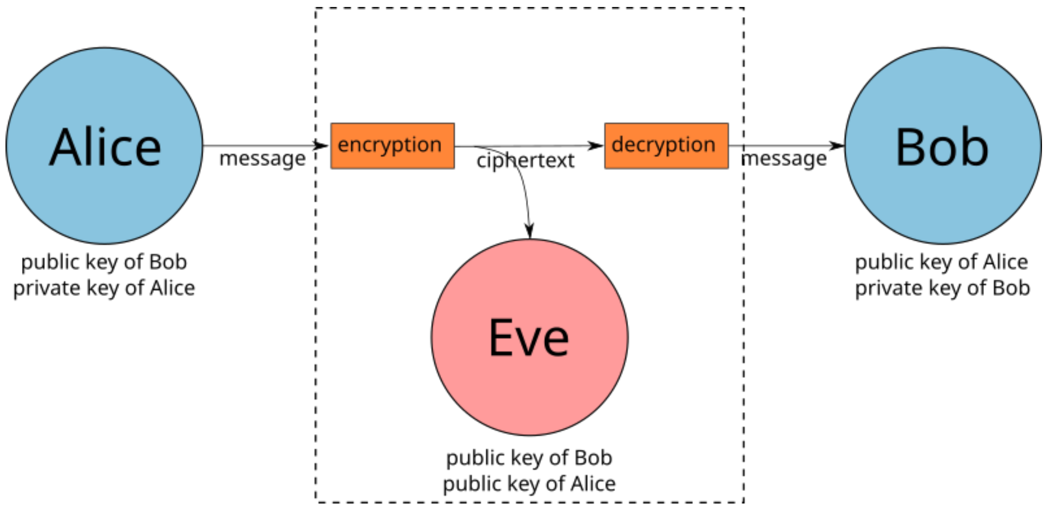


Figure 1.3: Public encryption scheme: Alice sends an encrypted message to Bob. The parties have already exchanged their public keys. Alice encrypts a message for Bob and sends it. Eve is some third party who obtains the ciphertext of the message and the public keys of Alice and Bob. Eve knows the algorithm used for encryption and decryption but since they don't have access to the private key, they can't decrypt the intercepted ciphertext. The attacker could encrypt their own message for Bob and send it.

1.2.1 The RSA cryptosystem

RSA is the one of the first cryptosystems realized to use the full potential of public-key cryptography. The name RSA comes from the initials of the creators Ronald Rivest, Adi Shamir and Leonard Adleman[6]. The system was created in 1977 and is today one of the most used cryptosystems.
The main use cases for RSA are[6]:

- Data encryption. RSA is generally used only for small data sizes because it's much slower than symmetric-key encryption, for example it's used for key transportation.

- Digital signatures. Systems which allow a recipient of a digital message or document to verify the authenticity of the data.

RSA is built on the mathematical problem of integer factorization. It is computationally easy to multiply two prime numbers, and it could also be done by hand on paper. But it is extremely hard to factorize the result and find the two prime numbers. This type of function is called one-way function because it's easy to calculate in one direction but it's extremely hard to reverse.

1.2.2 Elliptic Curve Cryptosystem

Elliptic curve cryptosystems (ECC) are the newest type of systems to create a public-key cryptosystem. Even though this approach exists since the 1980's, it's just lately that it got much more attention and interest. It offers significant advantages to other public-key cryptosystems. Elliptic Curve Cryptography provides the same level of security as RSA but with much shorter key length, which can significantly improve efficiency and reduce network traffic overhead.
RSA is constructed based on the difficulty of factoring prime numbers, on the other side elliptic curve cryptography is based on the difficulty of solving the

elliptic curve discrete logarithm problem[7]. The elliptic curve discrete logarithm problem is basically the problem of determining the number of steps needed to move from one point on a elliptic curve to a different point[7]. Even if the the equation for the curve and the start point is known, it's impossible to determine the number of hopes that were made between the start point and another point on the same curve[7].

Elliptic Curve Discrete Logarithm Problem. *Given two points P and Q where Q is a multiple of P. Find k such that $Q = kP$.*

The security of ECC is built on top of the difficulty of solving the above problem[7].

To better understand the "hopping" on a curve let's look at some char-

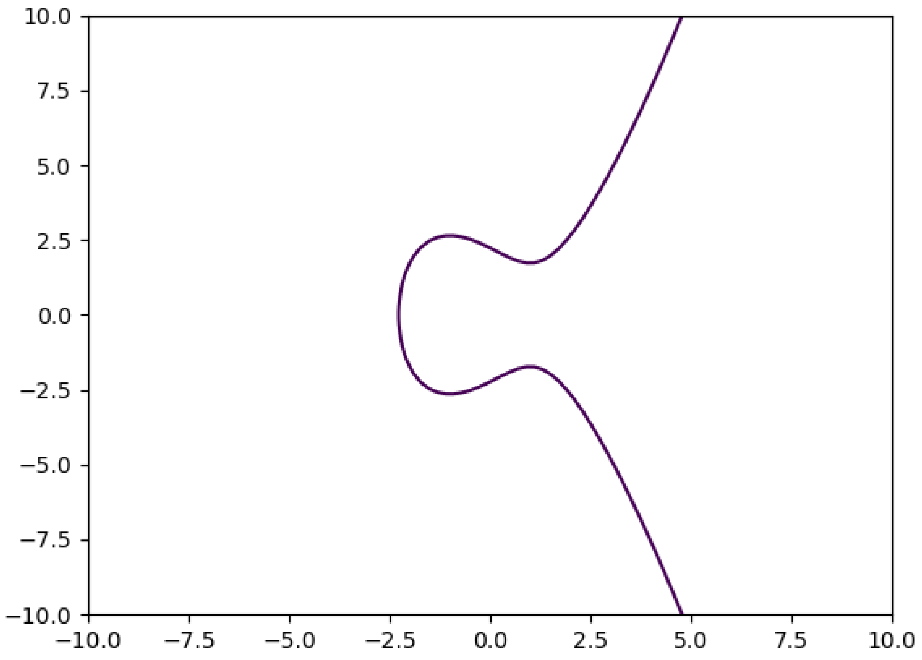


Figure 1.4: Elliptic curve $y^2 = x^3 - 3x + 5$

acteristics of an elliptic curve. An elliptic curve used in cryptography is a

plane algebraic curve in the form of $y^2 = x^3 + ax + b$, where x and y are the standard variables used like in any other algebraic function to represent points on a Cartesian coordinate plane. The coefficient values a and b transform the elliptic curve and define its specific appearance. An elliptic curve is symmetric in respect to the x-axis, and any non-vertical line intersects the curve in at most 3 points[7].
Figure 1.4 shows an elliptic curve on a Cartesian coordinate plane with the coefficient values $a = -3$ and $b = 5$.
Elliptic curve cryptography uses essentially two operations for hopping between points on an elliptic curve in order to find values used for encryption[7]:

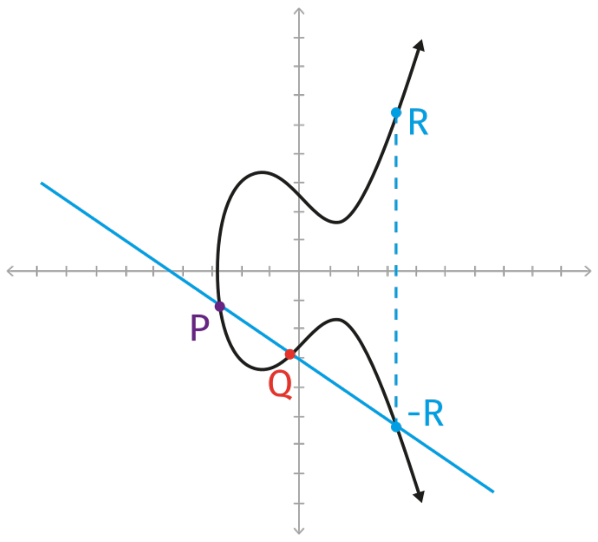


Figure 1.5: Example of Point Addition[7]

- Point Addition. This is an operation that allows to derive a new point starting from another point on an elliptic curve. Given two points P and Q, a third point is created by drawing a line that goes through the points P and Q, as shown in figure 1.5. This results in a newly formed point at the intersection between the line and the elliptic curve, that is called $-R$ [7]. Since an elliptic curve is symmetric to the x-axis, this also creates a point R, also shown in figure 1.5. By finding the point $-R$ it's simple to obtain the sum between P and Q, hence $P + Q = R$.

This progress then can be repeated by drawing a line crossing P and R which creates another point on the curve and finally by drawing a vertical line at this point a new point is found that is the sum between P and R[7].

- Point Doubling. This is another operation used in ECC. It works similar to Point Addition but instead of adding two points, the point P is added to itself[7]. Instead of drawing a line crossing two points, a tangent line to the point P is drawn, as shown in figure 1.6. This line then intersects the elliptic curve at some point $-R$. Finally a vertical line through the point $-R$ is created and the intersection with the elliptic curve gives the point R, hence $2P = R$ [7].

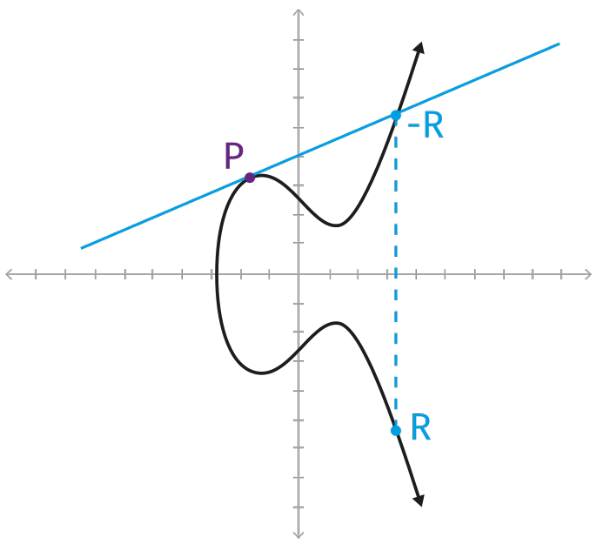


Figure 1.6: Example of Point Doubling[7]

In addition to Point Addition and Point Doubling, ECC also uses the concept of finite field[7]. It isn't possible to include every possible point found on an elliptic curve, therefore, the set of valid points is limited by a point called p. The point p represents the key size and by increasing it's value the security of the system can be increased[7].

1.3 Threshold Cryptosystems

A Threshold cryptosystem is a system that allows to encrypt an information with a public key. The private key is then distributed among all the participating entities, and no entity should be able to access the decryption key and therefore be able to obtain the original information (or any part of it) without the agreement of at least a predefined number of parties. In a system that has n parties and uses t as a threshold, the minimum number of parties that have to give consent, is called (t, n)-threshold. In a similar way it's also possible to create a signature scheme which requires multiple parties to sign a document. Many different public-key encryption schemes can be used to create a threshold cryptosystem.

1.3.1 Secret Sharing

Informally, secret sharing is any process that allows to split a secret into shares and to distribute the shares among a group of individuals called shareholders[8]. Such a system requires also that no single individual can retrieve the information without at least some number of other shares, this means that each share doesn't expose any information about the shared secret.
Generally the objective is to create (t, n)-threshold scheme, where a secret S is divided into n shares and given to n shareholders, so that the knowledge of at least t shares is needed to make the secret S easily retrievable, but fewer than t shares expose no information at all about S.
Secret sharing is useful for managing and securing cryptographic keys. It is simple to protect data by encrypting it, but for protecting encryption keys different systems are required. Secret sharing schemes try to address this issue. They are systems that are more resilient to data loss and more secure against security breaches because they distribute the secret over multiple individuals[9]. This implies that also its control is shared among more entities, hence making it more difficult for attackers to obtain the shared secret

because they need to obtain many different pieces from different parties.

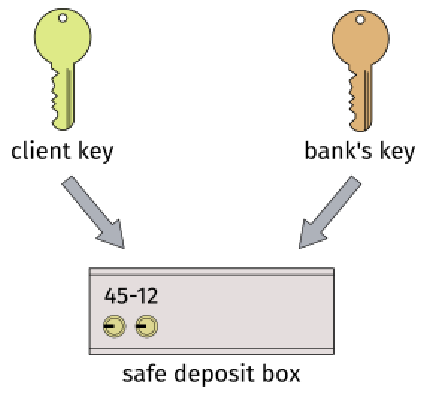


Figure 1.7: Safe deposit box. A $(2, 2)$-threshold scheme

An example of secret sharing is a safe deposit box in a bank. Normally they use two separate keys and both keys are needed to open the deposit box. One key is given to the client and the other one is kept by the bank manager. This way none of the two parties can access the content by themselves, thus both are required to open the box. This example creates a $(2, 2)$-threshold scheme, since it requires the two shares of both individuals to access the secret, in this case the deposit box.

Example of insecure secret sharing

Let the following system be a (4,4)-threshold scheme. The secret is $S =$ "*somesecuresecret*". The secret S is split in four equal parts and distributed to the four individuals as shown in table 1.1.

This system is faulty because each individual already knows one-fourth of the secret S. But a secure secret sharing scheme requires that no party can gain any knowledge about the secret without overcoming the threshold, in this case all four individuals are required.

The table 1.2 shows how much knowledge is gained based on the number of shares obtained. The third column shows the number of possible values an attacker would need to guess to retrieve the secret S. The objective in

S	some	secu	rese	cret
s_1	some			
s_2		secu		
s_3			rese	
s_4				cret

Table 1.1: Faulty Secret Sharing: pieces of each individual

shares	missing letters	possible values
0	16	26^{16}
1	12	26^{12}
2	8	26^{8}
3	4	26^{4}
4	0	26^{0}

Table 1.2: Faulty Secret Sharing: knowledge about the secret based on the number of shares[8]

a secret sharing scheme is that even when $t-1$ shares are known it's still impossible to know anything about the secret itself. In this example the desired outcome would be that with $t-1$ shares the possible values are still 26^{16}, but already with only two shares the number of unknown values is cut in half.

Trivial secret sharing

There are three types of secret sharing:

(t, n)-threshold scheme with t = 1 In the most trivial system of secret sharing, the secret can be shared directly among all parties, since every individual can access the secret without any consent from other parties.

(t, n)-threshold scheme with t = n There are many different schemes which allow to share a secret among n individuals where every share is required to retrieve the secret. Two examples of this type of systems are[8]:

- The secret is encoded as an integer S. Let's give to each participant i, out of the n participants, a random number r_i, excluding one participant k. Let's give to the shareholder k, who has not received any share so far, the number calculated by this formula $(S - r_1 - r_2 - ... - r_n - 1)$. To obtain the secret S it's simply a matter of summing up the shares of all shareholders.

- The secret is encoded as a byte S. Each shareholder i, excluding shareholder k, obtains a random byte b_i. The shareholder k receives the resulting byte of $(S \oplus r_1 \oplus r_2 \oplus ... \oplus r_i)$, where the $\oplus$ is the bitwise XOR. To retrieve the secret S a bitwise XOR of all shares is calculated.

(t, n)-threshold scheme with 1 < t < n This type of secret sharing is more complex than the other two, since the objective is to create a system that is secure but it doesn't require all shares to retrieve the secret. Also, this kind of system can't release any information at all about the secret without reaching the number of shareholders to overcome the threshold, otherwise it wouldn't be secure.
Consider a secret of a company that is shared among the members of the board of directors. The business doesn't want that one individual has full power over the company's secret, but it would also be unwise to require all shareholders to access the secret, because some of them could be unable to access (for example if they die) and therefore it would be impossible to access the secret ever again. A possible solution would be a ($\frac{n}{2}$, n)-threshold scheme where a majority is required to gain access to the secret.
It is possible to create this kind of scheme based on a (t, t)-threshold, like one of the above examples. The (t, t)-threshold scheme is used n times then t shares are distributed to each individual[8]. Since each individual is required to store multiple shares this solution isn't space efficient and there are better solutions for this problem.

The parameters t and n

The choices of the threshold t does heavily depend on the application as well as the number of individuals n. A secret sharing scheme is appropriate to be used in a situation where a group of mutually suspicious individuals with conflicting interests are required to cooperate[8]. This setting still needs a sufficient large majority to be able to perform an action or access the secret, but on the other hand it also requires to have a sufficient large minority that could block an action or deny access to the secret. The choice over different thresholds comes with a couple of trade-offs. By increasing the threshold t for a given number n of individuals it reduces the availability of the secret, the access to the shared information, but on the other side it also increases the secrecy of the shared information. The secrecy can be defined by the number of shareholders that an attacker is required to convince or compromise to give up their shares to gain access to the secret. The integrity of the scheme is set by the number of parties an attacker needs to corrupt at least to modify or change the secret, it's calculated by $n - t + 1$. Therefore, the selection of the threshold must be adjusted for different use cases as well as for different situations.

1.3.2 Shamir's Secret Sharing

Shamir's Secret Sharing is an efficient way to share a secret and implements a (k,n)-threshold scheme. The objective is to split some data S into n pieces $S_1, ...S_n$ so that it satisfies the following requirements[9]:

- the knowledge of any k or more S_i slices turns S easily computable.
- the knowledge of any $k - 1$ or fewer S_i slices leaves S completely undetermined, which means that all possible values for a secret have the same probability to be the real data S.

Shamir's Secret Sharing scheme is based on polynomial interpolation[9]. It would also be possible to use any other collection of functions that allows an easy evaluation and interpolation.

Theorem 1 (Interpolation)**.** *Given k distinct points $x_1, ..., x_k$ and the corresponding values $y_1, ..., y_k$, there exists a unique polynomial $q(x)$ of degree $k-1$ such that $q(x_i) = y_i$ and consequently interpolates the data $(x_1, y_1), ..., (x_k, y_k)$[9][10].*

A random $k-1$ degree polynomial is created in the form of $q(x) = a_0 + a_1 x + ... + a_{k-1}x^{k-1}$ where a_0 is set to S encoded as an integer, so $a_0 = S$. Without losing generality it's possible to assume that every data S can be represented as a number. To obtain a share for each shareholder $S_1 = q(1), ..., S_i = q(i), ..., S_n = q(n)$ is calculated.
Provided any subset of size k of the values S_i and their identifying indices i it's possible to find the coefficients of $q(x)$ by interpolation, hence gain access to the secret S by calculating $S = q(0)$. The knowledge of less than $k-1$ share, at the contrary, doesn't give enough information to calculate S.
Shamir's Secret Sharing scheme uses modular arithmetic instead of real arithmetic to make the calculation more precise. The system uses a set of integers modulo a prime number p, that is bigger than the integer S and n, this creates a field where interpolation is possible. Then the coefficients $a_1, ..., a_{k-1}$ in q(x) are randomly selected from a uniform distribution over the integers in the set $[0, p)$ as well as the values $S_1, ...S_n$ are retrieved modulo p.
Assuming that an attacker gains access to $k-1$ shares, for every possible value S' in $[0, p)$ they could create one unique polynomial $q'(x)$ of degree $k-1$ such that $q'(0) = S'$ and $q'(i) = S_i$ for the available $k-1$ arguments. By design all p possible polynomials have the same probability and therefore the opponent can't deduce any information about the real secret[9].
There are efficient algorithms which interpolate a polynomial through n points with complexity of $\mathcal{O}(n \log^2 n)$[11], but even with less efficient quadratic algorithms it would be acceptable fast to interpolate and to create a usable key management system[9]. In addition to its computational efficiency this type of system is also space efficient since no key is bigger in size then the

data S. Shares can't be arbitrarily short, the prime number p needs to be bigger than $n+1$ because the set $[0, p)$ needs to contain at least $n+1$ distinct values to calculate $q(x)$.
The following list summarizes some of the key features of the Secret Sharing scheme created by Shamir[9]:

- The size of each share doesn't exceed the size of the original data, by splitting the data into smaller pieces the size of the shares could be reduced.

- As long as k is fixed the scheme allows easily to add and remove shares dynamically. To remove a share the shareholder is required to completely give up the knowledge of the share.

- It's possible to generate a new set of shares S_i without changing the original secret S. This can be done by creating a new polynomial $q(x)$, which can significantly increase the security because an attacker would have to gain access to enough shares from the same generation to overcome the threshold in order to retrieve the secret S.

- A scheme created in this way has the ability to create a hierarchical structure by using tuples of shares where individuals with a higher power receive more shares and this way have more control over the secret. Imagine a board of directors of a company. The president gets three shares, the vice-president obtains two shares and other stakeholders get one share. By creating a $(3, n)$-threshold scheme the secret can be accessed by any three executives, by two executives when the vice-president is present, or the president by themselves.

1.4 End-to-end encryption

In online communication, end-to-end encryption is used to make sure that the conversation between two users can only be read by them[12]. Users can

be humans, but the same principles can be applied to machine communication as well.
Generally, a public-key cryptosystem is put in place to share the keys between the parties and may also be used to encrypt the content. Some systems use a private-key cryptosystem to improve performance and use an asynchronous system to share the encryption keys which then are used by both entities.
Communication systems which do not implement end-to-end encryption can be vulnerable to eavesdroppers. End-to-end encryption doesn't protect only against outside attackers but makes it also impossible for anybody who doesn't have access to the decryption keys to read the content. This reduced significantly the required trust the user must give to the provider of the communication system because they have no way to access the content of the conversation. Of course, the keys need to be stored in a secure way and can't be shared with third parties, which would impact significantly the security of the system. Many services that claim to fully encrypt messages send by users often use closed source implementations that are not possible to independently inspect and validate. Even worse they often store cryptographic keys on their server[12]. This puts the keys not only out of the hand of the user, but it also leaves the provider in full control over the keys, so anyone who compromised the security of the company would obtain the content of a conversation. Obviously if anybody would gain access to the keys used for the communication that entity could eavesdrop on the traffic. Furthermore, a well implemented e2e encryption system does obfuscate also the data from the internet service provider as well as from any server which forwards the communication to the receiving user.
Additionally, an end-to-end encryption system needs to provide a method to verify that the user receiving authorization (or the decryption keys) to read the message is actually the correct user. If there is no way to identify a user an attacker could still listen to the conversation by putting themselves in the middle of the traffic. The attacker would work as a relay who has two interaction points, one that talks to the real sender and another one that

forwards the messages to the real destination. The attacker would simply present themselves to be the receiver to the real sender and to be the sender to the receiver of the message. This form of intrusion is called man-in-the-middle attack.
End-to-end encryption still leaves the user unprotected from attacks that target directly the user's computer, eventually the invader could capture the message before it wasn't encrypted or when it was already decrypted. It could also be possible that an attacker steals the cryptographic keys from the user's machine and this way get access to the conversation.

1.4.1 The Double Ratchet Algorithm

The Double Ratchet Algorithm is a key management algorithm that together with other cryptographic protocols is used to provide end-to-end encryption for instant messaging. It was initially created for Signal, a fully encrypted instant messenger, by Trevor Perrin and Moxie Marlinspike in 2013[13]. Nowadays it's used by many different platforms like WhatsApp[14], Wire[15], Matrix[16] and many more.
The algorithm is called double ratchet because it combines a cryptographic ratchet based on the Diffie–Hellman key exchange and a ratchet based on a key derivation function (KDF) for example a hash function. Diffie–Hellman key exchange is a secure protocol for exchanging cryptography keys over a public communication channel.
The Double Ratchet Algorithm uses, after the initial key exchange, short-lived session keys which are constantly renewed. The algorithm is said to be self-healing because in certain situations the protocol inhibits an attacker who has gained access to session keys from reading the cleartext of the conversation after one message has passed the attacker without them compromising the message[17].

1.5 Hash function

A hash function is a one way function, a function that is easy to calculate in one direction but it's basically impossible or infeasible to invert, which maps a string with an arbitrary length to a string, often called hash or digest, of fixed length. This type of functions have many use cases and applications because they allow to convert a string from any length into a hash with fixed length where the output doesn't release any information about the input string. Hash functions are a cryptographic primitive used for digital signature, message authentication codes and for other forms of authentication. One of the most important features of a hash function is that they return always the same digit for the same input string, this property is used to proove the authenticity of data. It is often used in combination with public-key schemes which provide confidentiality of the message.
A simple and often used application of a hash function is to give a user the possibility to verify the integrity of a file or software downloaded from the Internet. The user calculates a hash value of the downloaded file on their own, and then they compare the generated value with the value obtained from the provider of the file.

To satisfy basic security requirements any cryptographic secure hash function needs to have the following properties:

- Determinism. Provided the same input string to a hash function, it is required that the same output hash is generated.
- Quick computation. The calculation of a hash, starting from any input string, needs to be fast.
- Collision resistance. It is required that it is highly unlikely that two different input strings generate the same hash as output. In other words, it needs to be difficult to find two messages which have the same digest.

- Non invertibility. A digest can't expose in any circumstances any information about the original message which was used to generate the digest.

- Avalanche effect. Every change, also only of one character, of the input needs to create a completely different digest.

Over the past year cryptographers have created a number of different hash functions, some of them have already known weaknesses and others seem secure enough for now and nobody could find an attack to compromise their security. A few of the most used hash functions which are standardized are:

- MD5. It was developed in 1992 by Ronald Rivest. The MD5 algorithm is designed to be calculated quickly and it returns a 128-bit message digest. This algorithm is obsolete nowadays since it's relatively easy to find hash collisions and therefore it shouldn't be used anymore, at least not for security relevant applications.

- SHA-1 (Secure Hash Algorithm 1) SHA-1 was created by the NSA and standardized by the National Institute of Standards and Technology (NIST) in 1993. Shortly after it was redrawn and in 1995 the now used version was published. This algorithm produces a 160 bits long digest. In 2017 it was shown by Google with the so called SHAttered attack that it it's possible to find two colliding messages that create the same digest, with a method which is significantly faster than a brute forcing a SHA-1 collision with a birthday attack[18].

- SHA-2 (Secure Hash Algorithm 2). SHA-2 was also created by the NSA. It has a similar internal structure to SHA-1. SHA-2 is divided into two distinct hash functions: SHA-256 and SHA-512. There exists a few other variants which are derived from the two mentioned functions. The number in the name represents the bit size of the output, e.g SHA-256 creates a 256 bit message digest.

- SHA-3 (Secure Hash Algorithm 3). SHA-3 was standardized and published by NIST in 2015. The SHA-3 is part of a broader cryptografic hash function family called Keccak. All functions in this family use a sponge construction, different from the Merkle–Damgård construction used by SHA-2, SHA-1 and MD5. And therefore, this family is resistant to the length extension attacks. Ethereum uses a slightly different version of SHA-3, called Keccak-256, which can create some confusion because the SHA-3 is sometimes used interchangeably with Keccak-256[19].

hash function	message digest	length
M5	946c a9b0 fcc3 5f84 678a dc2d f85e 43e8	128 bit
SHA-1	5e6c 2c9f 1116 ab3a 36bc e9c8 44ab 34b7 df3a a747	160 bit
SHA2-256	34d7 9f3c 1113 ad6d 6550 ccdb b750 3670 9a22 9107 55ec 2f4a d09e c551 d49c 3cdf	256 bit
SHA2-512	636e c24d b100 9600 480e 6902 da98 c9d3 b7a1 f7ef 754f 9368 0d37 5504 5b4c e272 77a5 051c b60d 3e1b 5941 6978 cb10 5c7f d33b 3783 6487 d07f a590 23ae e32a ec91	512 bit
SHA3-512	267a 7f7b 73ee cdef 8ef2 04a5 53be 51c5 6c5b a5ab 4388 993d ba5c 6d3f 7ea5 e68e 047f 2902 bb85 831f f619 2020 9e96 bc0a d8c7 14bb ca3a 50d7 39f8 6ccc fa6b abf8	512 bit
Keccak-256	0326 0d84 f984 f392 6310 92b4 89a1 13f9 2424 e08d 8afe 1fa4 b159 f759 b452 b309	256bit

Table 1.3: Comparison of message digest produced by different hash function for the same input string: ”lorumipsum”

Chapter 2

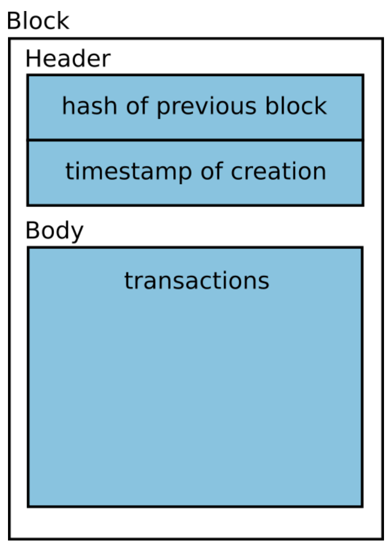
Blockchain

The blockchain has many different definitions, depending on who you ask the question you can receive a different answer because there is so much interest from different areas in blockchain technology. The blockchain emerged together with its first implementation Bitcoin. Bitcoin was created in 2008 by an unknown person or group named Satoshi Nakamoto[20]. The main purpose of Bitcoin is to create a decentralized digital currency. While Bitcoin is a cryptocurrency, the underlying blockchain technology was used to create a wide range of different systems. This chapter will focus mostly on use cases beyond its application in finances, since that isn't the scope of this thesis.

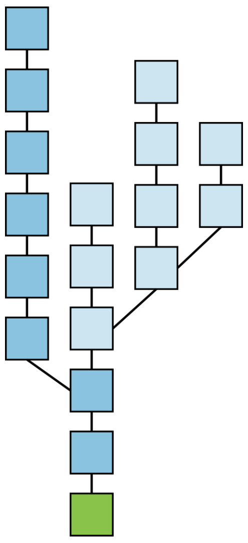
A blockchain is essentially a data structure that allows to store data in a non-structured manner. The information is distributed over a network of nodes. Records which store the user's data are called transactions. A transaction is inserted into a logical group for organization named block[21].

The name blockchain comes from how it operates: each block is linked to the previously created block, so in this way a single linked list or chain is created. Each block therefore requires a pointer to the previous block. Normally the pointer is a cryptographic hash of the former block. Each block can also contain more metadata, like the timestamp, but the full structure of a block depends on the specific design of the system.

The distributed nature of a blockchain also requires a distributed process to produce new blocks. Different blockchains have different ways to create them, and manage the flow of blocks' validation differently. Some of the specific methods are discussed later in this chapter. Each newly produced block is attached to the end of the chain by incorporating the hash of the previous block. This obviously can produce two blocks at the same time because usually there is no centralized authority, and when this happens a fork in the chain is introduced. The creator of a new block, typically, attaches it to the longest chain they know.



(a) A possible block structure in a blockchain



(b) A chain in a blockchain, the green block is the origin which doesn't have any parent. The main chain is shown in darker blue, and the blocks in light blue are shorter chains, that have split off from the main one.

Figure 2.1: An example of a block and created chains in a blockchain.

Some of the key features of a blockchain based system are:

- Immutability. Data added to the blockchain can't be changed. Even though it could be done with a huge amount of computing power, it is considered nearly impossible to modify data, therefore the blockchain is considered immutable[21].

- Transparency. Most blockchain implementations are open source, which allows contributions by anyone and it makes the system publicly auditable. Any transaction stored in a block is public and can be reviewed by every entity using the blockchain. This obviously reduces confidentiality and privacy, but many applications need to audit the flow of transactions in order to avoid the problem of double spending and other related problems.

- Distributed consensus. Blockchains use different mechanisms to reach an agreement among different nodes on how to create new blocks. Those methods are called distributed consensus. There are many different possible methods to create an agreement between nodes. Therefore, no single node has control and makes decision, but multiple parties are required to agree to give the consensus.

- Security. The blockchain technology is built on the top of a proven cryptographic primitive, like for example hash functions, to ensure the integrity of the data. The immutable nature of the data allows everyone to verify the information, as well as it's nearly impossible to manipulate the stored data.

- Smart contracts. Smart contracts are essentially code run on top of the blockchain. Not every blockchain implements smart contracts, but because of its versatility it's a much-desired feature[21].

2.1 Centralized, decentralized and distributed systems

There are three major topologies regarding how networks can be structured. This chapter will explain the main properties and the differences between them.
First let's define what a network is in the context of this work. A network is a system with several nodes that are in some way connected and can communicate. Each node is one unit interacting with other nodes in the system. A node can be for example a physical device.

- Centralized system. Each node communicates exclusively with the central node, in figure 2.2 the green node is the central point. It's the simplest structure, because it doesn't require any complex system for consent, or to resolve inconsistencies of the system's states. This type is the most used system because of its simplicity.

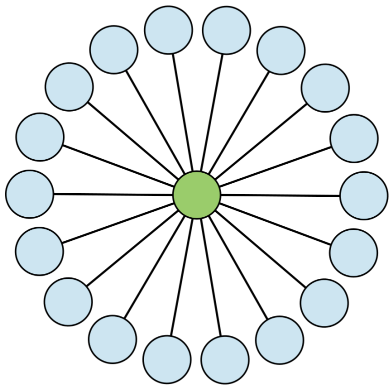


Figure 2.2: A centralized system. Each circle is a node, while the green circle is the central server node. Every client node interacts only with the central node.

A centralized system is often referred to as a client-server system. Normally the server provides a service, for example it can offer a webpage.

Each node, other than the server, is a client and it doesn't interact with anybody other than the server. It's still possible to allow communication among multiple nodes but the traffic must always be relayed by the central node. Generally, all the data are stored on the central server, and users update the system's memory or state via requests to the server.
Centralized systems give a great control over the data flow, as well as over the entire systems, since it has by design a central authority. The centralization also comes with downsides, the architecture is much more prone to system failures and it comes with many privacy issues.

- Decentralized system. This type of system is quite different from a centralized system. A decentralized system has multiple central points, shown as green circles in figure 2.3. This topology requires a more complicated process to keep the system consistent, since no single point has the full power over the network. Also, it needs sophisticated processes for consent and to establish an agreement between different nodes.

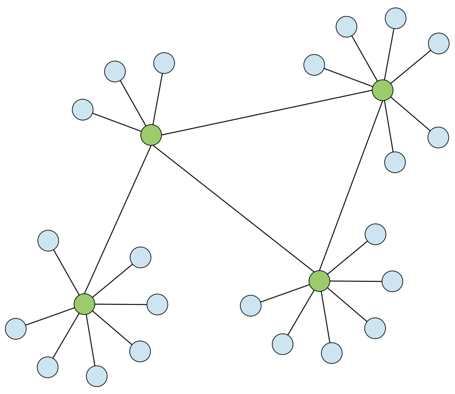


Figure 2.3: A decentralized system. There are multiple central nodes, shown as green circles. The blue circles are the client nodes.

The decentralized ownership and control is an advantage and it improves the resilience of the system as well as it makes it more difficult for an attacker to take over the entire network. Therefore, by design, decentralized systems resist much better to system failures and attacks. Also, they increase the privacy of their users because no single entity is in power to read all the traffic on the network, at least not from a single point.

- Distributed System. A system is called distributed when a collection of independent nodes, generally computers, act autonomously to complete a task[22]. Normally a distributed system is intended to appear as a single unit to the user. To accomplish this, it uses complex methods to manage the system and to distribute tasks.

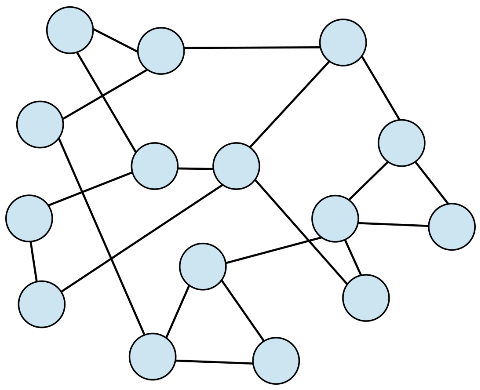


Figure 2.4: A distributed system

Ethereum and Matrix, used in this work, are both decentralized systems by design. They were built to remove the central entity and to give the ownership of the network to the people running and using the network. Ethereum, or in general blockchains, are distributed networks and this is explained in more detail in the section about smart contracts.

2.2 Merkle tree

Hash digits are used to provide integrity checks for transactions stored in a block. To make the calculation of hash digits and to allow fast verification of transactions, a blockchain uses a particular data structure called Merkle tree. The Merkle tree was named after Ralph Merkle who patented it in 1979[23]. It's a binary tree where every leaf node is labeled with the hash of the transaction data and all non-leaf nodes are labeled with the hash of the label of its immediate children. This layout allows to determine if a leaf node is part of a hash tree by calculating a few hashes proportional to the logarithm of the number of nodes in the tree[24]. In a blockchain it's vital to verify the validity of a block or transaction, therefore the Merkle tree is used. Figure 2.5 shows a Merkle tree with four leaf nodes and the label of each node.

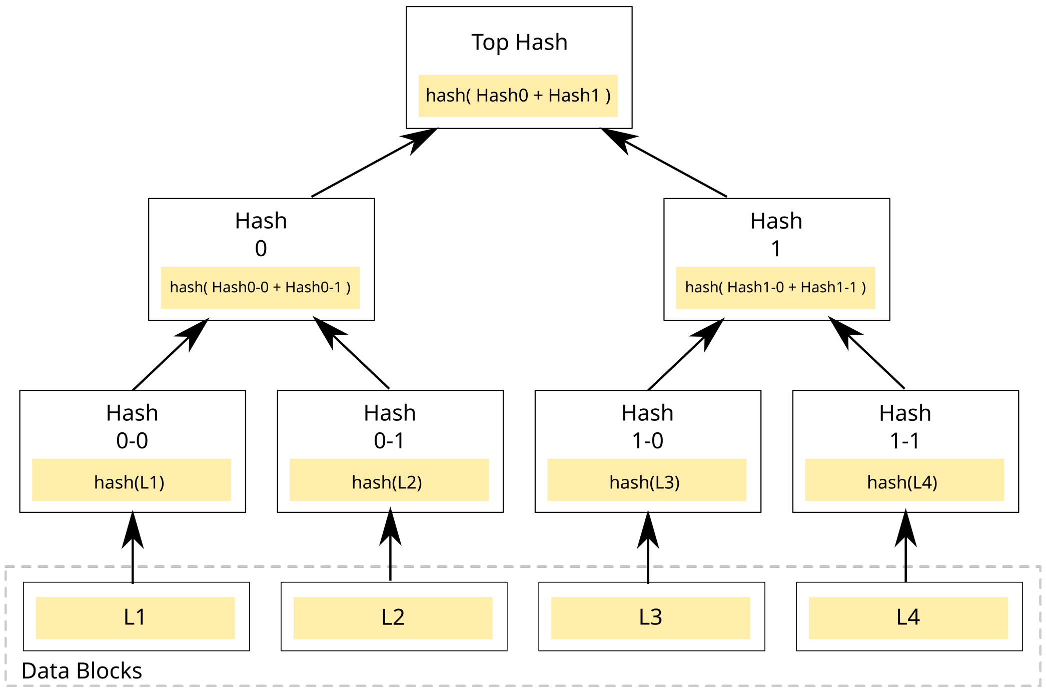


Figure 2.5: Merkle tree. A binary tree with four leaves and a depth of four. Highlighted in yellow are the labels of each node where L1, L2, L3, L4 are the hash of the transaction stored in the block. Each non-leaf node is labeled with the hash of the concatenation of the labels of its direct children[25].

2.3 Blocks

A node listens for new transactions and adds them to the list of transactions to be inserted in a new block. Every node in the system can generate new blocks. A blockchain is a distributed system and consequently there isn't any central authority to control when a block is created and who can create it. Evidently, it's not possible to just randomly create new blocks without the agreement of other nodes. Therefore, a blockchain puts in place different methods for creating new blocks. This process allows autonomous nodes to agree on something in a distributed manner. In other words, there are different consensus mechanisms to select a new block. Some of the consensus mechanisms are reported later in this chapter.

A newly created block contains the cryptographic hash of the previous block

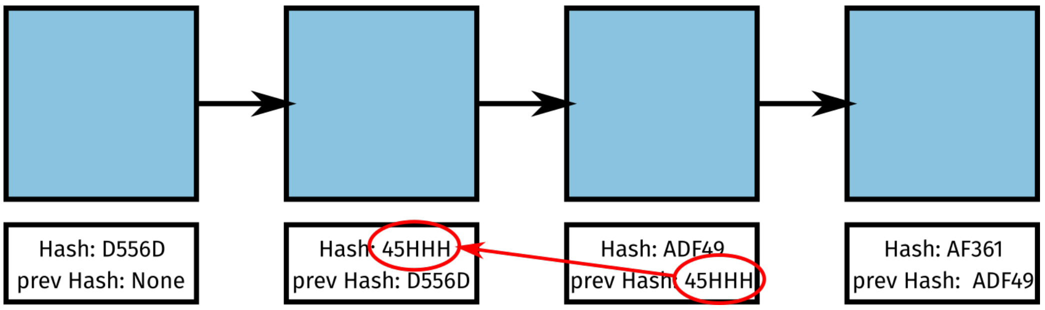


Figure 2.6: A sequence of blocks building a chain. Each block has a reference of the previous block via the cryptographic hash. The block on the far left has no previous block, hence it's the first block ever created.

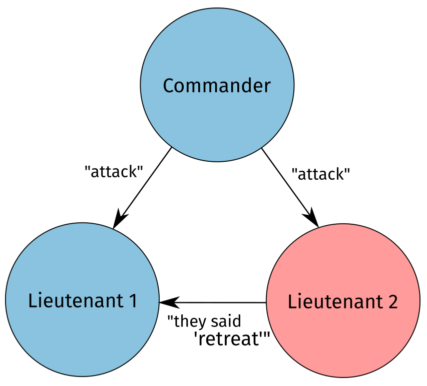
(shown in figure 2.6), as well as the transactions the node decided to include into the block, hence the transaction needs to exist before the new block is started to be created. By appending new blocks to the chain, a single linked list is formed, where the hash is the link. Because the system is distributed, more than one block can be produced at the same time by different nodes. When this happens the chain is forked, that means that it is divided into two chains. In this way a tree-like structure, shown in figure 2.1, is formed. Other nodes then usually continue attaching new blocks to the longest chain.

Naturally blocks of shorter chains are dropped at some point and the transactions stored in those blocks are lost if not included in other blocks.
If the system becomes out-of-sync it could also be possible that some nodes continue generating blocks for a different chain, and at some later point they overwrite the previous predominant chain. By quickly generating new blocks it's possible to obtain the same effect and in this way manipulate data stored in the blockchain. This from of attack is mitigated by slowing down the block creation as well as making block generation as expensive as possible (computational and economically) via the consensus mechanisms.

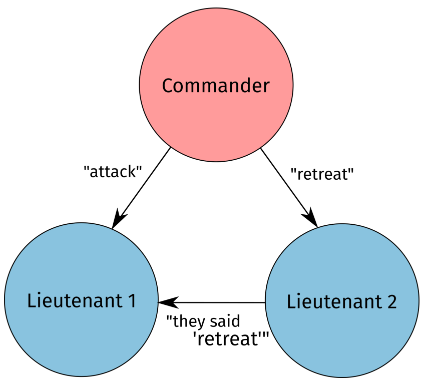
2.3.1 Consensus mechanisms

The problem that a consensus mechanism tries to solve was formalized long before the emerging of the first blockchain. And it was researched in the context of distributed computing. To better visualize the issue a thought experiment was created: The Byzantine Generals problem [26]. The problem consists of a group of army generals who command different parts of the Byzantine army. The generals need to agree on whether to attack a city or to retreat, since in order to conquer the city and to win the war the full power of the Byzantine army is needed. The only method the commanders can use to communicate is a messenger. Unfortunately, some of the generals can be traitors and send intentional misleading messages to other generals[21].

Let's consider for an example three generals, shown in figure 2.7. Each general is in control of one third of the Byzantine army. The generals are divided into a Commander and two Lieutenant, they all are equal and have the same power, but the Commander sends the first message. In figure 2.7a the "Lieutenant 2" has a malicious intent, therefore they invert the message they get from the "Commander". In this way the "Lieutenant 1" doesn't know if they should attack the city or retreat. In figure 2.7b the commander themself is the traitor and communicates two different messages to the other two generals and thus "Lieutenant 2" sends a different message



(a) The Lieutenant 2, in red, is a traitor.



(b) The Commander, in red, is a traitor.

Figure 2.7: Byzantine Generals problem. The Commander sends the first message to the Lieutenants and then Lieutenant 2 forwards the message to the other Lieutenant.

to the ”Lieutenant 1”. Again ”Lieutenant 1” is confused because they get two contradicting messages.

In this thought experiment it’s only possible to gain an agreement between the generals if more than one third of the generals are trustworthy, therefore in the above example one traitor is enough to disrupt the entire system. Thanks to modern cryptography it’s possible to build systems that perform much better. Protocols that sign messages can obtain a consent even when more than one third of the participants are traitors.

The Byzantine General Problem can be seen as an analogy to a distributed system like the blockchain. Each general is a node and the messenger are a communication channel between the nodes.

A consensus mechanism is a protocol executed by every node, or most nodes, in the network to create an agreement on some decision. Blockchains can use different consensus mechanisms to create new blocks. The most dominated processes are Proof of Work and Proof of Stake.

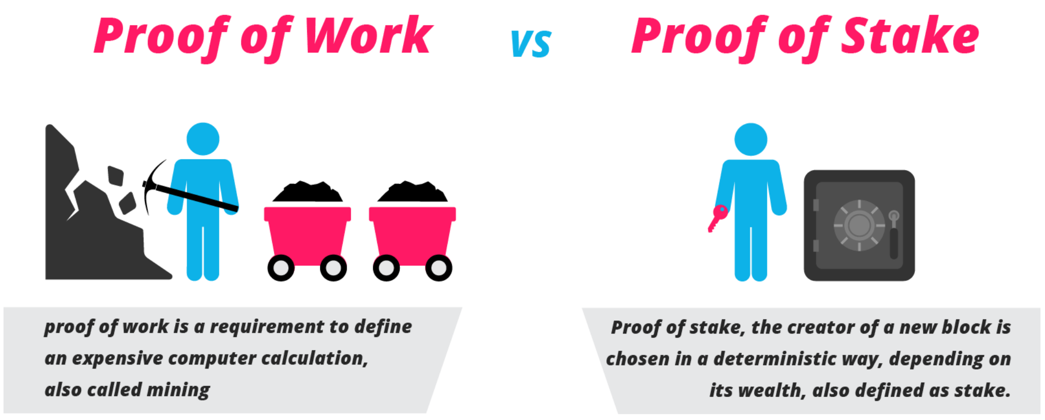

Proof of Work
vs
Proof of Stake
proof of work is a requirement to define an expensive computer calculation, also called mining
Proof of stake, the creator of a new block is chosen in a deterministic way, depending on its wealth, also defined as stake.

Figure 2.8: Proof of Work vs Proof of Stake [27]

Proof of Work

Proof of Work relies on the proof that enough computational resources were used to obtain the proposed value. This type of consensus mechanism is used in many cryptocurrencies and it's the most used mechanism for the validation of a block. In a blockchain network, different nodes compete to solve a mathematical puzzle.

This process is called mining and the actors doing the heavy computation are called miners. The first miner to solve the mathematical problem distributes their block to the rest of the network and hence they confirm all transactions present in the new block. The miner who created the block receives a compensation generally in two different forms, via transaction fees paid by the initiator of the transaction, and depending on the blockchain, also some currency. In Bitcoin for example they receive a specific amount of bitcoins, and this amount is constantly reduced based on the number of blocks generated overall. Therefore, at some point in the future miners will only receive the transaction fees. The difficulty of the puzzle is adjusted to the overall performance of the network to stabilize the time passing between each newly created block. This consensus mechanism has a couple of downsides. If an attacker owns more the 50% of the computing power of the system, they can insert a malicious block[27]. Once a block is found by a miner all other miner's effort to solve the puzzle is lost. Therefore, miners often combine their computing power into mining pools to try to solve the problem together faster than others. This can significantly reduce the decentralization of the network.

Since solving a puzzle is computationally very expensive, the calculation

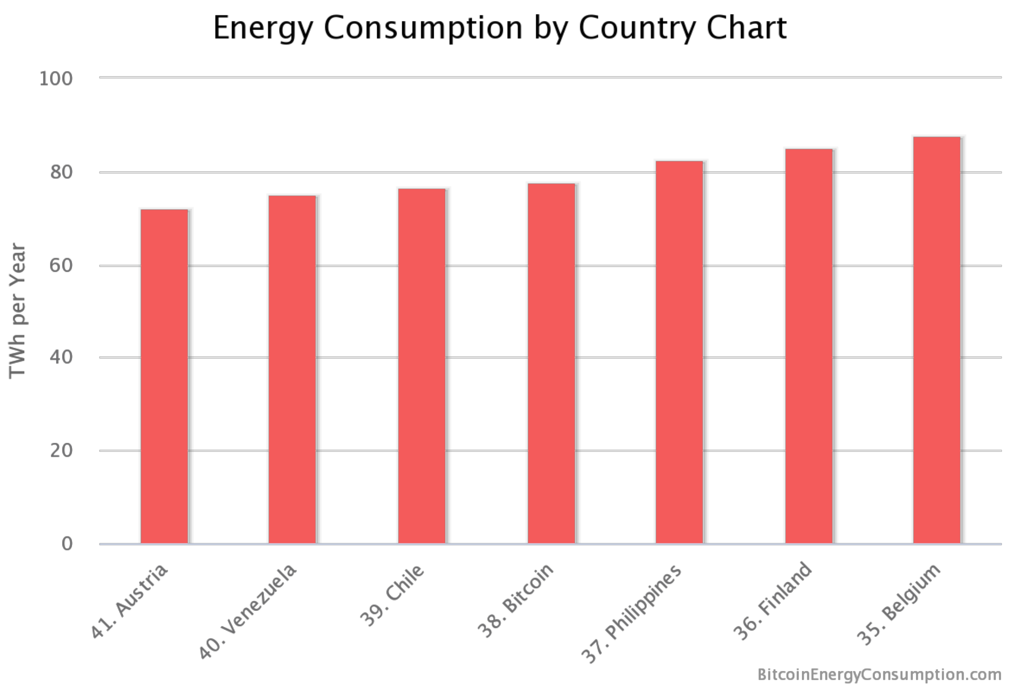


Figure 2.9: Energy consumption of the bitcoin network by country[28]

requires huge amounts of electricity to run the infrastructure. The bitcoin network consumes 77.78 TWh in one year, which is as much electricity as the entire country of Chile uses in one year[28]. A single transaction consumes more electricity than an average household in the US in about 23.45 days[28]. If the network would be a country it would place itself as the 38th biggest energy consumer in the world.

Proof of Stake

Because of the high energy consumption of Proof of Work researchers invested a lot of time to create a different consensus mechanism that doesn't

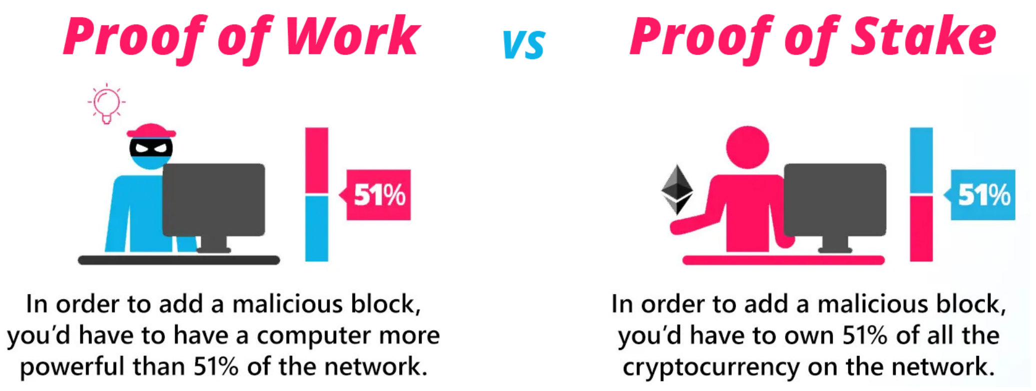

Proof of Work
vs
Proof of Stake
51%
51%
In order to add a malicious block, you'd have to have a computer more powerful than 51% of the network.
In order to add a malicious block, you'd have to own 51% of all the cryptocurrency on the network.

Figure 2.10: Proof of Work vs Proof of Stake. Network dominance.[27]

waste as much resources. Obviously without weakening the security of the system. Unlike PoW, Proof of Stake is based on the wealth of the user for creating a new block. A user who wants to create a new block must deposit some amount of cryptocurrencies called stake. Then a user is selected randomly where users with a higher stake have an increased probability to be selected. This works because if the user misbehaves during the consensus process, they lose the money in the deposit. If they behave correctly, they will receive a compensation of the work, the transaction fee of each transaction included into the block. This implies that user with a bigger stake are more trustworthy.
Naturally this creates a fundamental problem. Richer users can risk more money and therefore also create more blocks and so get even richer. This issue is often described with the expression ”the rich get richer”. Another problem is also that if a single user controls more than 50% of the wealth of the blockchain they can created malicious blocks, but this problem exists in a similar form for Proof of Work as well.
A slightly different variant of Proof of Stake used by PeerCoin [29] adds the factor of age. The money has an age which gets reset every time the resource is used to validate a block. Therefore, the user must wait a specific time to reuse the same money to create a new block.
Proof of Stake significantly reduces the energy consumption of the system

and hence it is much more environmentally friendly. Also, the energy consumption of a single transaction is pretty much negligible. Some people argue that Proof of Stake is a more secure mechanism then Proof of Work because it's more complicated to obtain a significant amount of cryptocurrency then it is to invest to build computational powerful computers for mining new blocks.
Ethereum is planning to perform a hard fork of the system to move from the current system based on Proof of Work to Proof of Stake.

Delegated Proof of Stake

Delegated Proof of Stake was first introduced with the blockchain Bitshares [30]. It is similar to Proof of Stake but stakeholders don't create blocks themself but delegate the creation to trusted nodes. Each user obtains a specific number of votes proportional to the assets they hold, they then elect key figures which are specially trusted users. This type of systems generally uses two different roles: ”witnesses” and ”delegates”. However not in every implementation they are strictly different in the tasks the execute. In Bitshares a witness performs the job of creating and validating new blocks . The delegate's chore is to oversee the correct functioning of the system, and they can adjust parameters of the system if needed. They also set the amount of compensation a witness receives for creating new blocks. If a witness doesn't behave well they can be excluded from future elections or won't get any compensation for a block they created.

Proof of Authority

This type of consensus mechanism is similar to the Proof of Stake. The trust isn't based on the stake a specific user holds, but on the authority the user has. The so-called validators, who have the power of inserting transaction into new blocks, aren't elected instead they earn the position. This system works on the principle that each user who gained the ability to validate and create blocks has the desire to maintain this position and therefore

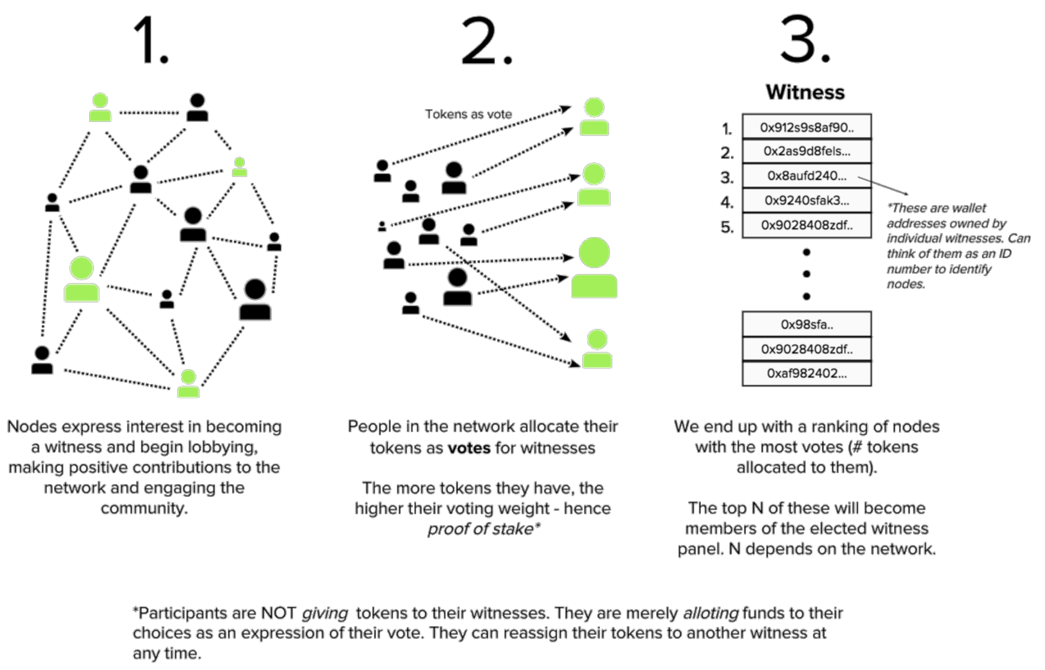

1.
2.
3.
Tokens as vote
Witness
1. 0x912s9s8af90..
2. 0x2as9d8fels...
3. 0x8aufd240...
4. 0x9240sfak3...
5. 0x9028408zdf..
0x98sfa..
0x9028408zdf..
0xaf982402...
*These are wallet addresses owned by individual witnesses. Can think of them as an ID number to identify nodes.
Nodes express interest in becoming a witness and begin lobbying, making positive contributions to the network and engaging the community.
People in the network allocate their tokens as votes for witnesses
The more tokens they have, the higher their voting weight - hence proof of stake*
We end up with a ranking of nodes with the most votes (# tokens allocated to them).
The top N of these will become members of the elected witness panel. N depends on the network.
*Participants are NOT giving tokens to their witnesses. They are merely alloting funds to their choices as an expression of their vote. They can reassign their tokens to another witness at any time.

Figure 2.11: Electing witnesses in a Delegated Proof-of-Stake network[31]

it assumes that the user acts in a manner to keep a clean reputation. Especially since their identity is well known.
This type of protocols is considered generally more efficient than other consensus mechanisms but it present a significant drawback. Since only a few nodes can create new blocks they are in control of the network. Therefore, the system results to be less decentralized. For this reason, this type of mechanism is more prevalent in private blockchains. Another frequently disused issue concerns the public identity of the validators, which makes it easier for third parties to manipulate and corrupt the system.

2.4 Smart contracts

Smart contracts were initially proposed a long time before the emerging of the first blockchain. They gained a lot of its popularity thanks to the blockchain technology. Even though the name smart contract leads to the assumption that it's just about contracts in the conventional sense, smart

contracts are much more than that.
A smart contract is a computer program that is executed on a distributed network of computers. They don't require any external authority to establish trust among mutually distrusting nodes running the code[32]. Smart contracts allow to define contractual clauses that make them in parts or fully self-executing or/and self-enforcing. It's important to implement smart contracts by using verified design patterns to create truly trustless and self-executing code especially since coding errors can lead to fatal damages. The infamous DAO attack is an example of an incident where an attacker was able to steal $\sim 50M$ USD because of a programming error[33]. A project aimed at helping developers to avoid this kind of bugs is OpenZeppelin[34], it provides tooling to create secure smart contracts, as well as a certified process for auditing the code.
Not every blockchain incorporates this feature, but it gained a lot of attention lately and it is a much desired feature and therefore many blockchains have an advanced infrastructure to use them.
Bitcoin implements smart contracts in a relative simple fashion and it allows only to write code in a non-Turing complete scripting language, hence it's limited in its use. On the other hand, Ethereum, used in this work, has a much more advanced system for smart contracts using a Turing-complete language. The smart contracts in Ethereum are explained in the next chapter.

Chapter 3

Ethereum

Ethereum is one of the most popular and most used blockchain-based systems. It was initially formalized by Vitalik Buterin in 2013[21]. Ethereum's key difference compared to Bitcoin is the focus on smart contracts. Bitcoin uses a limited scripting language for writing smart contracts and hence it allows only basic operations. Ethereum on the other hand uses a Turing-complete language for smart contracts, this allows to write arbitrary programs and run them on the blockchain in a distributed fashion. Therefore, Ethereum is much more a distributed computation platform than a cryptocurrency.

Ethereum uses Ether as its integrated currency. It is used to reward users for mining new blocks. Ethereum implements a consensus mechanism based on the Proof-of-Work mechanism also used in Bitcoin. Ethereum has an interesting approach to transaction fees. Fees are payed in the form of Gas. Gas is fundamentally the fuel needed to execute an operation. The amount of Gas needed for a transaction is based on how much computational, storage, network, etc. a specific computation requires. For example, a function in a smart contract that performs a task requires a specific amount of Gas to execute.

A distributed app (Dapp) is a fancy way to interact with a smart contract or with multiple contracts[35]. A Dapp is essentially a webapp which runs its back-end logic on Ethereum. Dapps aren't just limited to Ethereum or to blockchain-based systems and they exist also for other distributed networks but were mostly popularized by the blockchain technology.

3.1 Network stack

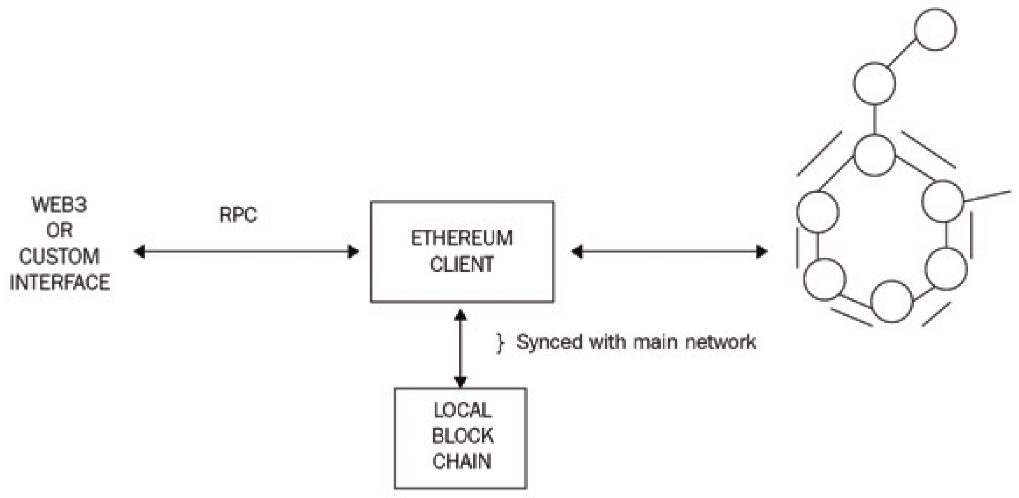


Figure 3.1: Simplified network stack of Ethereum[21]

Ethereum can be seen as a transaction-based state machine[21]. Each transaction is stored as a change of state. By applying every transaction to a genesis state, the initial system state, the current state of the system can be retrieved.
Ethereum uses a peer-to-peer network as the foundation for communication between blockchain nodes. Each Ethereum client running on a computer forms a node of the network. One physical machine can run multiple clients and therefore create multiple nodes. Each node synchronizes themself with other nodes over a peer-to-peer network. In this way a node creates their own copy of the blockchain. Normally an Ethereum client provides a remote procedure call (RPC) API, that allows a user to interact with the node.
Ethereum uses a consensus mechanism based on the protocol Greedy Heaviest

Observed Subtree (GHOST)[36]. It was created by Zohar and Sompolinsky in 2013. Bitcoin gives the longest chain the precedence. Ethereum on the other side uses the most expensive, in sense of computational power, as the dominant chain.
An important concept of Ethereum are accounts. The overall system state is defined by the account state. Each operation between accounts modify the account state. An account is identified via a public hash with length 160 bit (20 bytes) called address.
There exist different Ethereum networks. The network types are generally divided into public, private and test networks.

3.2 Currency

Name	Value
wei	1 wei
kwei, ada, femtoether	10^3 wei
mwei, babbage, picoether	10^6 wei
gwei, shannon, nanoether, nano	10^9 wei
szabo, microether, micro	10^{12} wei
finney, milliether, milli	10^{15} wei
ether	10^{18} wei
kether, grand, einstein	10^{21} wei
mether	10^{24} wei
gether	10^{27} wei
tether	10^{30} wei

Table 3.1: The Ether denominations[37]

Ethereum uses Ether as its currency. The table 3.1 shows a list of the denominations of the different units used in Ethereum. Users receive Ether as reward for the computational power they invest for mining new blocks.

To execute a smart contract a user spends Ether to buy Gas. Gas can be considered as the fuel needed to process a request. The required Gas for an execution is estimated based on storage and network requirements.
The transaction fee for an operation is calculated via this formula $GasPrice * usedGas$. The miner who includes the transaction in a block receives the fee as compensation. In the previous formula the $GasPrice$ is set by the originator of the transaction and is the amount the user likes to spend for an operation. The $usedGas$ is the actual amount of computation needed to execute the operation. The originator sets the Gas limit when they initiated the transaction. If the execution of the transaction requires more than the Gas limit, then the execution is stopped, and state is rolled back to the state it was before staring the transaction.

3.3 Smart Contracts

Smart contracts in Ethereum are written in a Turing-complete programming language. A Turing-complete programming language allows to simulate a Turing machine. In other words, it allows to write arbitrary programs and to build new instructions based on the available instruction set. This makes Ethereum a platform to execute any code in a distributed way.

3.3.1 Ethereum virtual machine (EVM)

The Ethereum virtual machine is used to execute the code of a smart contract. The EVM is a simple stack-based-execution machine, which modifies the state of the system by executing a smart contract[21]. A smart contract is translated by compiling into bytecode which then is interpreted by the virtual machine. Ethereum's virtual machine is a Turing-complete machine, but an execution is limited by the Gas consumed. The Gas limitation is used to eliminate the problem of infinite loops which could block the entire network. The EVM creates a fully isolated and sandboxed runtime environment which doesn't allow any access to external resources like the network or the host filesystem[21]. Each operation performed by the EVM has a specific Gas cost. For example, the calculation of a SHA3 hash costs 30 Gas, and the creation of a new contract costs 53000 Gas[21].

3.3.2 Solidity

There are multiple languages a user can choose to implement a smart contract, but the most used programming language is Solidity. Solidity is an object-oriented, high-level language [38]. It was built specifically for the purpose of implementing smart contracts. Solidity is a statically typed programming language. The language was inspired by languages like C++, Python and JavaScript and therefore includes many features known from other popular languages. Solidity is compiled to bytecode and is then executed by the EVM.
The following code example is a smart contract written in Solidity. The first line defines the minimum version of Solidity. Then the name of the contract is defined, which contains three variables and two functions. The function *set_secret* set the variable *secret* of type bytes32 to the value provided by the caller of the function. The function also sets the variable *secret_is_set* of type bool to *true*. The *require* keyword is used to state the requirement for the execution of the function, when the statement isn't satisfied no operation is executed. Thanks to the *require* the secret can be set only once. The second function *get_secret* returns the secret to the caller, but only if the sender is the owner that was set previously.

```
pragma solidity ^0.5.0;
contract SampleContract {
 bytes32 secret;
 address owner;
 bool secret_is_set;

 function set_secret(byte32 _secret) public {
   require(secret_is_set == false);
```

```
    secret = _secret;
    secret_is_set = true;
    owner = msg.sender;
  }

  function get_secret() public view returns (byte32) {
     require(msg.sender == owner && secret_is_set ==
        true);
     return secret;
   }
}
```

3.4 Ethereum clients

Ethereum has several different clients that allow a user to create a node, hence create a blockchain network. The client software contains an interface that allows to interact with the node and the network.
The best known Ethereum clients are Geth and OpenEthereum (Parity). Geth is written in the programming language Golang. Geth is sometimes used as the underlying client for other nodes applications. For example, the Ethereum client Mist, a user-friendly Ethereum client[21], uses Geth under the hood.
OpenEhtereum is another client written in the programming language Rust. Geth and OpenEthereum expose both a Remote Procedure Call (RPC) API, which allows to interact with the Ethereum node.

3.4.1 OpenEthereum (Parity)

OpenEthereum is one of the fastest and most advanced clients for Ethereum. It recently changed its name from Parity to OpenEthereum, therefore in this document the names may be used interchangeably. The entire

Figure 3.2: OpenEthereum's logo (Parity's logo)[3].

codebase is licensed under the GPLv3, and hence it is free and open source software[3]. OpenEthereum is built with the objective to create a secure, fast, and lightweight Ethereum client, therefore it uses the emerging programming language Rust. Rust comes with many safety guarantees which are mission critical for a secure Ethereum client.
Key features of OpenEthereum include[3]:

- OpenEthereum is built to be customizable, therefore it focuses on a clean and modular codebase.
- The client has an advanced command line interface.
- Thanks to a feature called Warp Sync the synchronization time is significantly reduced.

OpenEthereum has some technologies built directly into the client that other clients don't incorporate. This work is built on top of OpenEthereum's Secret Store core technology[4].

3.4.2 Secret Store

The Secret Store is a technology integrated into OpenEthereum that allows to securely store encryption keys on the Ethereum blockchain. This

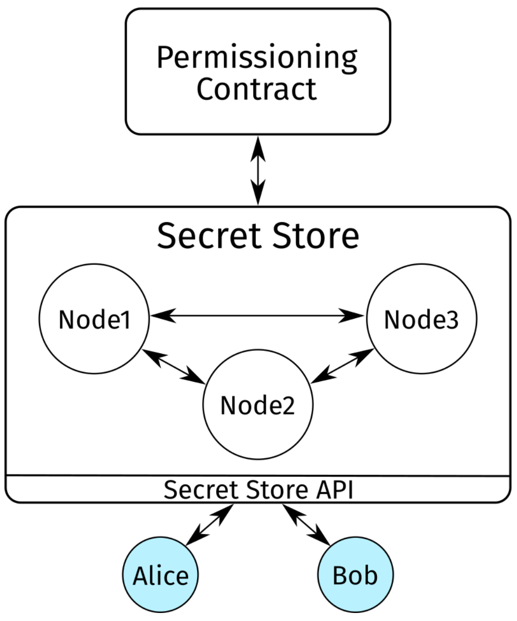


Figure 3.3: A Secret Store setup composed of three nodes.

feature can be used to encrypt a document as well as to sign a document. Since the cryptographic keys are distributed over multiple shares which are stored across multiple nodes a distributed key storage is created.

The Secret Store implements a public-key cryptosystem based on elliptic curve cryptography. The elliptic curve cryptography was used to minimize the size of the cryptographic keys without compromising security[4]. The key generation and the key storage was implemented based on the paper "ECDKG: A Distributed Key Generation Protocol Based on Elliptic Curve Discrete Logarithm[1]"[4]. Additional, the Elliptic Curve Digital Signature Algorithm (ECDSA) used to sign documents was implemented based on the paper "A robust threshold elliptic curve digital signature providing a new verifiable secret sharing scheme[2]"[4].

The Secret Store uses a dedicated interface to interact with other Ethereum

[1]ECDKG: A Distributed Key Generation Protocol Based on Elliptic Curve Discrete Logarithm: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.124.4128rank=1

[2]A robust threshold elliptic curve digital signature providing a new verifiable secret sharing scheme: https://ieeexplore.ieee.org/document/1562272

nodes, hence each node needs to run OpenEthereum and it is required to enable the Secret Store feature.
To control access permission to stored cryptographic keys the Secret Store uses a smart contract that, thanks to Solidity, allows to create a powerful permission system.
The user of the Secret Store can interact with it via a HTTP interface, which allows to generate the public key as well as the private key for the encryption of a document.
In this system the private key is never distributed to all parties and therefore it remains unknown to each single node, but each node receives a share of the private key, and by combining the shares the key can be retrieved. Same as for all public-key encryption systems, the public key can be distributed without limitations and restrictions. The access to the private key is gained when a threshold number of nodes give permission to access it. This forms a (t, n)-threshold system, where t is the number of shares required to access the secret, in this case the private key and n is the total number of nodes forming the Secret Store. In addition, also $t + 1$ nodes need to agree on the permissioning contract state[4].

Chapter 4

Matrix

Matrix is an open standard for real-time communication over IP[39]. Its main use case is instant messaging, but the protocol is intended to be usable for any type of real-time communication. It can also be used to drive Internet of Things, VoIP/WebRTC signalling and many other real-time applications, as long as the message can be expressed as JSON.
Matrix being an open standard means that the entire API is public and well documented[1]. The open nature of the protocol allows it to be easily extensible. Matrix was created with the ability to integrate with other communication systems, which is one of the core features of the protocol. Matrix uses HTTP[2] as the transport protocol and JSON[3] as message format[40].
In Matrix, an instant messaging conversation takes the form of a virtual room, each message sent to a room is represented as an event. Not only messages are considered to be events, but any other user change is also sent in the form of an event.
The entire system was designed to be decentralized and to operate without a central point. Each user connects to a server, called ”homeserver”, each server can be home to multiple users, but each user could also operate their own server. A user's server communicates with other homeservers to

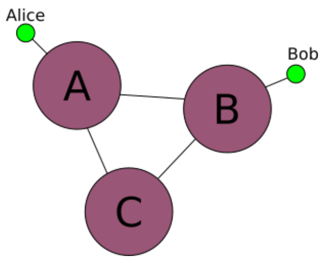
[1]Matrix Specification: https://matrix.org/docs/spec
[2]HTTP: https://www.w3.org/Protocols
[3]JSON: https://json.org

synchronize messages in a room.

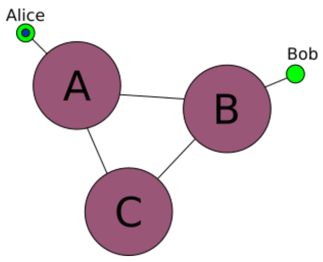
4.1 Architecture

Matrix is basically a decentralized conversation store[1]. The protocol has no single point of control or failure, if a room is distributed over multiple servers. A room can also have no distributed copies when all participants are on the same server.
In order to participate in a conversation a user needs to create an account on a server called their homeserver. The user is bound to the specific server and the server has total self-sovereignty over its users' data. A user can choose their homeserver from a wide range of different servers, most of them free of charge, but they can also run their own server and add it to the federated network.
The homeserver stores and provides account information and keeps the room history of rooms the user is a member of. A homeserver synchronizes the state and message history of a room with other homeservers, but only with servers which have a user participating in the room. In this sense Matrix works similarly to how commits are replicated and distributed over different git clones[1]. Whenever a user's homeserver goes offline, the communication for the users on that specific server is interrupted but the rest of the network won't be affected. Once the server turns back online it will synchronize its room state and message history, fetching it from other homeservers who participated in the conversation. In the same way as it's possible that a server goes offline, it's also possible that a server goes out of sync with other homeservers. A possible reason for this can be that there are too many users on the machine, or a slow internet connection. Matrix was designed to resolve this type of issue.

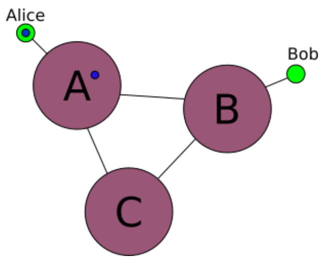
The figures 4.1 illustrate how an event (e.g. a message) flows from one user, through the matrix network, to a different user connected to a different



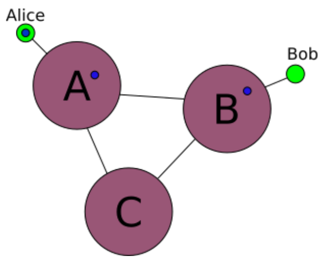
(a) No events



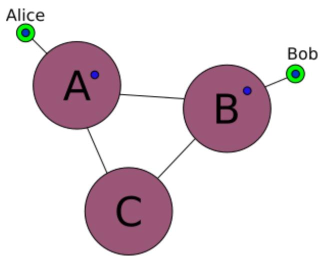
(b) Alice created an event, but didn't send it yet



(c) Alice has sent the event to their homeserver



(d) The event was propagated to Bob's homeserver



(e) Bob receives the message from his homeserver

Figure 4.1: Matrix network with 3 homeservers: A, B and C, and two clients: Alice connected to A and Bob connected to B. The black circle is a message event.

homeserver. The shown network contains three homeservers, A, B and C. The server A and B, have one user each, Alice and Bob respectively. For simplicity this example considers only one event and only one room. At first, figure a), there are no events. In figure b) Alice generates an event, but didn't send it to their homeserver yet. In figure c) the event is sent to server A, then in figure d) the event is propagated to the server B. Lastly in figure e) the user Bob receives the event.
Observing the process, it's possible to see that server C isn't involved at all in the conversation, since it doesn't have any user participating.

4.1.1 Users and Identity

A user is associated to a unique user ID, namespaced with the domain of the homeserver and is in form of *@localpart:domain*. A user ID can also be linked to a third party identifier, e.g. an e-mail address or phone number. The user ID is often called simply Matrix ID.

4.1.2 Events

Matrix packs all data exchanged over Matrix into an "event"[41]. An event normally represents one user action for example sending a message. Each event has a associated type, each type has to have a globally unique name. For example a message send to a room has type *m.room.message*. Not only data explicitly send by the user are expressed as a event but also room state changes are expressed the same way.
All events send to a room are stored in a directed acyclic graph (DAG) called "event graph". The chronological ordering of events in a room is given by the partial ordering of the DAG. Each event can have multiple successors, because multiple servers can race against each other to insert a new event in the room history. If the event doesn't have any parent it means that the event doesn't have any previous events known to the homeserver.

The following JSON is a message send to a room:

```json
{
  "type": "m.room.message",
  "sender": "@jsparber:gnome.org",
  "content": {
    "msgtype": "m.text",
    "body": "Hi all"
  },
  "origin_server_ts": 1585839155013,
  "event_id": "$DCsyYK3E9fK6QFHooj66GtX-nDe8hWGo3RYu2t4MMSI",
  "room_id": "!sYjxRUYKlgujLcpRun:gnome.org"
}
```

This object contains information about the sender, the type of message, the ID of the room it was sent to and an identification number for the event itself.

4.1.3 Rooms

A room is an abstract concept where users can send and receive events[41]. Each participant in a room with sufficient access create or read events. A room is a unique identifier in the form of *!opaque_id:domain*. The Room ID contains a domain which is only used for namespacing, but it doesn't mean that the room is stored only on the specific domain. Room events are split into two separate groups: *message events* and *state events*.

Message events

Message events are generally sent by the user directly. They can be, for example, an instant message or a file transfer. These types of event are part of the room history.

Room state events

State events describe updates to persistent information associated with a room, the state of the room. Possible properties include the room's name, topic, or membership. The state of a room is calculated by considering all preceding state events, which results in the room state at a specific moment in time.
A room can also have multiple aliases for identification. A *Room Alias* looks like *room_alias:domain*. The room alias makes it easier for users to find the room. This is especially important for public rooms where anybody can join.

4.1.4 Devices

The Matrix protocol has a special meaning for devices[41]. A device doesn't represent one physical device, but each client used. For example, a user could have multiple clients on the same physical device. Devices are mostly used to identify different clients and to generate client specific keys for end to end encryption. They are also useful to give users more control over connected sessions and logged-in clients.

4.2 End-to-end encryption

Matrix has built-in end-to-end encryption[16]. The encryption used in Matrix is based on the Olm and Megolm cryptographic ratchets[16]. Olm is an implementation of the double cryptographic ratchet based on the double ratchet specification[4]. The Double Ratchet Algorithm was initially created for Signal[5] (a fully encrypted instant messenger) but today it's used by other applications as well. Olm doesn't work very well to encrypt messages for large groups, which is why Matrix uses Megolm to mitigate these issues. Megolm is an AES-based cryptographic ratchet[42]. It still requires a differ-

[4]The Double Ratchet Algorithm: https://signal.org/docs/specifications/doubleratchet
[5]Signal: https://signal.org

ent encryption system, e.g. Olm, to setup some keys to work.
Currently the e2e encryption is still in beta. Therefore it's not enabled for all rooms by default, but it can be enabled on a per-room basis by users who have the necessary room permissions to do so. Once the encryption is enabled for a room it can't be disabled again. It will be enabled by default for all private rooms once it's out of beta. Public rooms on the other hand, have much less reason to be encrypted since anybody can join at any given point of time, therefore they probably will never be encrypted by default.

Device verification

Each of a Matrix user's devices is identified by a device-specific cryptographic key. A user has to verify another user's device by comparing the device-specific keys with the keys obtained from the other user. The exchange of the device keys can be done via any communication channel, as long as it allows a user to identify the other user. Possible channels include direct human interaction, or a voice/video call. It doesn't need to be an encrypted channel, because even when the communication is intercepted a attacker, the attacker won't be able to read any communication encrypted later. Since the encryption system is still in beta, the device verification will probably be subject to future changes[2].

Message encryption

Like everything else in Matrix, encrypted messages are also sent via events. There exists a special event type for encrypted messages called *m.room.encrypted*. Encrypted events contain two main properties: *algorithm* and *ciphertext*. The property *algorithm* identifies the encryption algorithm used to encrypt the ciphertext included in the event. The chipertext is the encrypted message event. The event can also include additional properties depending on the encryption algorithm used.

The following JSON shows an *m.room.encrypted* event:

```
{
  "type": "m.room.encrypted",
  "sender": "@jsparber:gnome.org",
  "content": {
    "algorithm": "m.megolm.v1.aes-sha2",
    "sender_key": "CdRdP0pWb3xiD+UUrAF/m0TSS8G5
       gQWioOCIWgVJEVc",
    "ciphertext": "AwgAEoABWZ4  s9qk6bVl4Ih7h09KkmkI44
       pSn+b//8xYJP1CFO62R+mvMzt3p8sBh4+isTV1Ge22AC3/
       pBRtvbyDiS3CbulwxD9pHiEj9WfpWbXorGtoD+xQNOy96
       jHhIpWkgx+hbvgQYZTXG6jLCZVqpmyNGcCQl4gFhN5YGqL6
       G/aJYPXm1x0kBT7JlNTuO0S3HWBCZXoYbTLG++rr4
       DDverYDac+FQznosxFuukff0rfxZDlL8fseofY5aJw8
       vtXJdg9IACS13fuLBdg4",
    "session_id": "LivTnCKQozahn5ntHwd9W3bSBH7
       XhlLXeajfjf940vE",
    "device_id": "JZGYMVYTNZ"
  },
  "origin_server_ts": 1585839197008,
  "event_id": "$AcgbVFcoICgYgO_iz3tlWTJnID5QvgWrrnIwH2
     lQuv8",
  "room_id": "!sYjxRUYKlgujLcpRun:gnome.org"
}
```

The following JSON is the decrypted payload of the previous JSON of a *m.room.encrypted* event.

```
{
  "room_id": "!sYjxRUYKlgujLcpRun:gnome.org",
  "type": "m.room.message",
  "content": {
    "msgtype": "m.text",
    "body": "Hi all"
  }
}
```

Chapter 5

Proposed design

An end-to-end encryption system needs to address a series of problems in order to be functional and secure. The encountered issues aren't purely technical, because the system needs to be simple and usable. The process needs to be easy enough that even people without any technical knowledge can use it, of course without compromising their security. To include non-technical users, the proposed end-to-end encryption system was designed with them in mind and the system was built to be as simple as possible to use.
The proposed system uses Matrix for the communication between users and implements message encryption via Ethereum.
This section discusses the design of the system used to encrypt and decrypt messages using OpenEthereum's Secret Store, and it describes possible solutions for key sharing and user verification. This chapter also explains the smart contract used as permissioning contract for the Secret Store.

The proposed system based on the Secret Store has the advantage that cryptographic keys are stored on the blockchain and therefore no backups of the keys are required because the blockchain already stores them in a distributed and redundant manner. Thanks to the smart contract used as permissioning contract the system can be extended with an advanced access control process to control the access to the conversation. It also allows to giving or revoking access to a message at a later point.

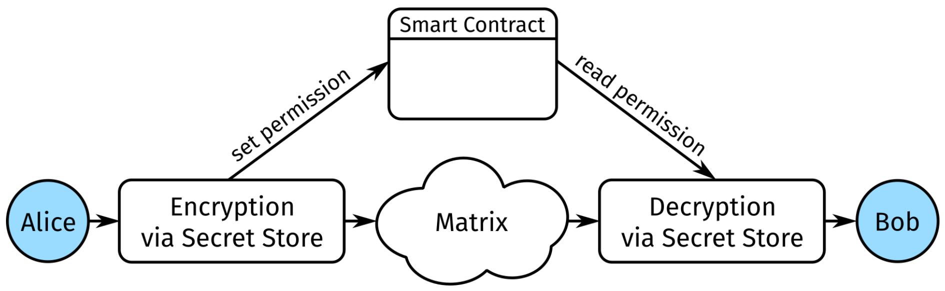


Figure 5.1: Abstract process of sending a message between two users: Alice and Bob.

In addition to the possible advanced permissioning contract, this system creates an abstraction from the cryptographic primitives in use as well as from the Secret Store itself, thus the system allows a developer to implement end-to-end encryption without much knowledge about cryptography nor about the blockchain technology. The created abstraction makes a clear distinction between the transport protocol used to send a message and the encryption. This property allows it to be reused for many different transport protocols, even those not designed for instant messaging or real-time communication.

5.1 Ethereum

The created system uses an Ethereum network to store the public and private keys. More specifically the keys are stored using the core technology of OpenEthereum called Secret Store.
The Secret Store is composed of a number of Ethereum nodes where the number of nodes is the upper limit of the usable threshold. The threshold is the number of nodes that need to agree on the permissioning contract state to grant permission to the private keys the allow an user to decrypt a message.
Each user who wants to use this end-to-end encryption system needs access to a OpenEthereum node and is required to create an Ethereum account. This Ethereum node should ideally run on the user's machine together with

the Matrix client. This node could be part of the secret store but it's not a hard requirement for the system to function. Of course, the process of setting up the Ethereum node and the account creation can be automated.

5.1.1 Permissioning contract

The secret store uses a smart contract to store the access permissions. It is written in Solidity which then is compiled into machine code that runs on the Ethereum virtual machine. The contract consists of two core pieces.

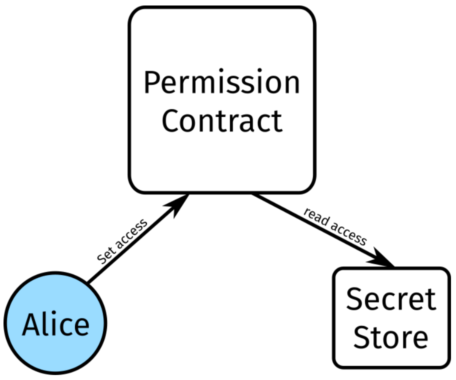


Figure 5.2: Interaction with the permissioning contract.

First, the creator of a message needs to set a list of users that are allowed to gain access to the decryption keys. Then the Secret Store can use the smart contract to check if a user is permitted to access them.
The secret store calls a function *checkPermissions()* to check the permissions. If this function returns *true*, the user is granted access to the stored key. Otherwise the access is denied. A certain number of nodes need to agree on the contract's state in order for this to work. The required number of nodes is set when a message is encrypted via a threshold property.
The smart contract was designed to be efficient, by reducing the number of calls required and by using memory efficient data structures, but to be still easy to read and to understand. Possible modifications and improvements are discussed later in this chapter.

Storage

To store information, the smart contract uses a special data structure called *mapping*. A *mapping* is a data type that associates a *key* of a certain data type to a *value* of any data type. This creates an optimal structure to link an input value to an output value. For example an account address to a message id. A feature of this data type is that any possible key is always associated with a value. If the value for a key wasn't set manually the value is automatically created and initialized with the default value of the data type when it is first accessed. This feature can be an advantage but can also be a disadvantage because it doesn't allow iteration over the entire set, at least not without exploring all possible keys. This is highly inefficient. However, since looping over a big dataset isn't needed in this case, this isn't a problem here. The proposed smart contract makes heavy use of *mapping*, because of its space efficiency.

The permissioning contract uses a *mapping* from message ids to a struct

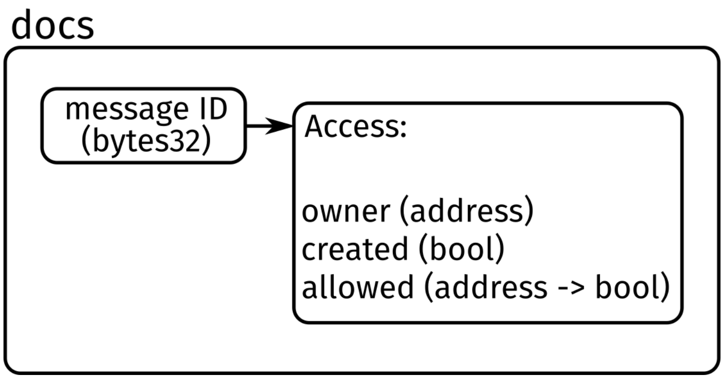


Figure 5.3: Layout of memory used in the permissioning contract.

called "Access". The struct *Access* contains three properties. The first property is the *address* of the owner, the *address* of the account which created the message. The second is a *bool* value which is set to *true* when the message is actually created. This property is required because mappings are assumed to always exist, and therefore any message id, even those not created by anyone, does exist. The property will be set to *true* only when the message is actually created by a user. The last property in the struct is a *mapping* of account addresses to a *bool* value called ”allowed”. The *mapping* from an *address* is set to *true* only when the owner of the message allows an user with the specific *address* to decrypt the message. Thankfully the data type *bool* uses as default value *false*.

Check permission

The method used by the Secret Store to check if an user is allowed to access the decryption keys called *checkPermissions()* requires two arguments, the address of user trying to decrypt a message and the message id (or often also called document id). The message id needs to be the same that was specified during encryption and creation of the message. The procedure returns a *bool*. It returns *true* if the user is the owner of the message, to guarantee that the owner always has access to their own messages, or if the creator gave the user previously permission to access the message.

Set permission

In addition to reading of the permissions from the smart contract, the contract also requires a method to manipulate the stored data. The contract uses a public function to give the creator of a new message the possibility to specify the access permissions.
The method named *allow_access()* is used to set the correct access permissions for a new message. It can also be used to update the permission for already existing message. The function takes two arguments. It requires the message id and a list of user addresses. The execution of the function is only allowed if the message wasn't created already or if the caller is the owner of the message. If it's a new message then the flag *created* is set to *true* and the sender is set as the owner. If the caller isn't the owner of the message then the function call is blocked and no memory is altered. Finally if the execution wasn't blocked so fare the list of user addresses is used to set the values in the *allowed* mapping to *true* for addresses in the list.

Full smart contract

This is the full smart contract written in Solidity which is used by the system.

```solidity
pragma solidity ^0.5.0;

contract SSPermissions {
  struct Access {
    address owner;
    bool created;
    mapping (address => bool) allowed;
  }

  // The main storage for messages, a mapping from
     document id to the struct Access
  mapping (bytes32 => Access) docs;

  // Function to give other users access to the
     decryption keys for a specific doccument
  function allow_access(bytes32 id, address[] calldata
      users) external {
    // If the requirement isn't satisfied the function
        call fails with an error
    require(
        msg.sender == docs[id].owner || docs[id].
            created == false,
        "Sender not authorized."
        );

    // Set sender to be the owner if it's a new
       document
    if (docs[id].created == false) docs[id].owner =
      msg.sender;

    // Set the right access permission for users
    for (uint i = 0; i < users.length; i++) {
      docs[id].allowed[users[i]] = true;
    }
  }

  // Function used by the Secret Store to check permission
  function checkPermissions(address user, bytes32 id) public view returns (bool) {
    if (docs[id].allowed[user] == true || docs[id].owner == user) return true;
    return false;
  }
}
```

Possible improvements

There are few possible improvements that could be made to increase the performance and also some features which where not included in the current version of the smart contract.
The smart contract for now doesn't allow to revoke permission, so if a user account gets compromised the owner of the message has no possibility to remove the user form the list of allowed users. This feature could be added via a function that takes a list of account addresses which is used to set the allowed property to *false*.
A achievable optimization to increase the performance of the smart contract could be to remove the loop in the function *allow_access()*. This function obviously is linear and its time consumption is proportional to the number

of users allowed to read a message. A possible solution to this issue could be to create groups of users which then can be allowed or denied in bulk. A new group of users could be created for each conversation so that the creator of a message only needs to give permission to a single entity. Each group could be represented by a different smart contract, that then can be used to reference a specific group. A system created in this way would be similar to the user permissions on an Unix-like operating system (e.g. GNU/Linux).

5.2 Matrix

Matrix's openness allows it to be easily extended and it gives the possibility to add features through custom events. This is one of the reasons why Matrix was selected for this work. It builds the foundation for sending messages. This project doesn't require any changes to the Matrix network or to the software running on the server. The client software, on the other hand, needs to be modified to handle the additional events used for this encryption system. The client will also require future changes, to give a user control over additional preferences and to provide feedback related to the security system.
Matrix uses JSON as format for messages and events therefore also the modified events use the same format. The end-to-end encryption system consists of two phases: The initial setup for a room and the ongoing encryption of messages.

Initial room setup

Matrix by default doesn't enable end-to-end encryption for all rooms. Each room has a room state which can be modified by sending room state events. The room state is essentially the set of preference and settings for a room. To enable end-to-end encryption the event *m.room.encryption* needs to be sent. This event defines the encryption method used. The event *m.room.encryption* contains the following values:

- algorithm. The algorithm used in this room. The default matrix end-to-end encryption allows two different algorithm. The proposed system adds an additional algorithm called ”secretstore.v1”.
- rotation_period_ms. This is the time after which the keys are renewed. Since in the proposed system new keys are generated every time a message is send this property isn’t used.
- rotation_period_msgs. This is the number of message send with using the same session keys, once the limit is reached new keys are generated. This isn’t used in the proposed system for the same reason as the previous property.

An example of the state event sent to enable encryption via the Secret Store is the following:

```
{
  "type": "m.room.encryption",
  "room_id": "!NAvrpHYMPTwEPpROkQ:gnome.org",
  "sender": "@jsparber:gnome.org",
  "content": {
    "algorithm": "secretstore.v1"
  },
  "origin_server_ts": 1585842810292,
  "unsigned": {
    "age": 5522477762
  },
  "event_id": "$IjGV4bhtqkNCubZHneX9_IqPsOU-n21
     UcKmCVojaY00",
  "user_id": "@jsparber:gnome.org",
  "age": 5522477762
}
```

Every property other then the *content* property are metadata including the sender and the room.

Message encryption

Matrix uses a special message event called *m.room.encrypted* to send encrypted messages. This event is essentially a wrapper around other events that are encrypted and attached as payload. The proposed design uses the same event as the default end-to-end encryption process.
An encrypted event has the following properties:

- algorithm. This is the algorithm used to encrypted this event. The list of algorithms was extended with the new method based on the Secret Store. The added algorithm is named ”secretstore.v1”.
- ciphertext. This is the payload of the event. It contains the encrypted event.
- sender_key. The key of the sender. This isn't used by the proposed design.
- device_id. A key identifying the device used to send the event. This event isn't used in the proposed design.
- session_id. The id of the session in use, this is only used in Megolm and isn't used in the proposed design.

The following example shows the JSON of a message sent to a room where messages are encrypted via the new system:

```
{
  "type": "m.room.encrypted",
  "room_id": "!NAvrpHYMPTwEPpR0kQ:gnome.org",
  "sender": "@jsparber:gnome.org",
  "content": {
    "algorithm": "secretstore.v1",
    "ciphertext": "AwgAEnACgAkLmt6qF84IK++J7UDH2Za1
       YVchHyprqTqsg"
  },
  "origin_server_ts": 1585844364450,
  "unsigned": {
    "age": 5520923604
  },
  "event_id": "$4Z3BhTSQ5w3QmLgzKOKn3tbVzkl0uq1
     ofpmhy_U7oME",
  "user_id": "@jsparber:gnome.org",
  "age": 5520923604
}
```

The property *ciphertext* contains the payload of the message. All properties except the *content* aren't specific to the encrypted event and are used also for other events.

The decrypted ciphertext looks like the following JSON:

```
{
  "type": "m.room.message",
  "content": {
    "msgtype": "m.text",
    "body": "Hello World"
  }
}
```

5.3 Identity verification

So far only message encryption was discussed. There is a second part to a working end-to-end encryption system. The identity verification can be seen as the process of verifying that the other party is actually the person or machine we want to talk to. Without verification there can't be any guarantee about the security of the system, because it would be easy for any attacker to impersonate a different user. A possible and simple attack would be a man-in-the-middle attack, where an advisory forges the identity of the com-

municating users and places themself in the middle of the communication.
Identity verification is often also refered to key verification since a cryptographic key is used to identify an user.
Overall there are basically two approaches for key verification. A public list linking encryption keys to users identities (for example Matrix user ids). This method doesn't require much user interaction and most of the verification can be automated. This needs some sort of mechanism which allows the owner to testify for it's identity. A good example for this method is KeyBase[1], which links social media identities to encryption keys. KeyBase is mostly know for its support of PGP keys[2] but it support also other encryption systems.
The other mechanism is to give the user the ability to check the key manually at the beginning of a conversation. This gives the sending party much more control about who they trust. Matrix uses this system for key and device verification. The planned way to verify the user is based on the second version described previously, but nothing would prevent a mixed system of being used.
The default encryption system used by Matrix, based on the double ratchet algorithm, allows an user to manually verify the device key to validate the other user's identity.
In the proposed system all encryption keys are stored securely on in the Secret Store, therefore the user doesn't have direct access to them. This system doesn't need key verification because the access to a key is strictly linked to a user via a the permissioning contract. However the user still needs to verify the Ethereum Account to make sure that the conversation is made with the correct party.
Some of the possible solutions to the previously described problem are:

[1]Keybase: https://keybase.io
[2]OpenPGP: https://www.openpgp.org/

Manual user verification

A different approach is to use a similar system to what Matrix already uses. It would be required to be modified so that it allows users to verify Ethereum accounts instead of cryptografic keys. This system requires user interaction to verify the identity.

Public list via smart contract

It's possible to use the blockchain to store a list of Matrix ids with the respective Ethereum account. To lookup an an account's address a call to a smart contract is made. The following contract uses a mapping to link the hash of the username to the address of the Ethereum Account.

```
 pragma solidity ^0.5.0;

contract AddressSharing {
  struct User {
    address ethaddr;
    bool created;
  }

  mapping (byte32 => User) users;

  // Takes the username as a 32byte hash
  function set_user(byte32 username) public {
    require(
        users[username].created == false,
        "User already registered."
        );
    users[username].ethaddr = msg.sender;
    users[username].created = true;
  }

  // Takes the username as a 32byte hash
  function get_user(byte32 username) public view
     returns (address) {
    return users[username];
  }
}
```

The advantage of this system is that when the correct information is stored it doesn't require any user interaction and the identity is guaranteed by the blockchain. This contract has the obvious drawback that whoever set the address for a Matrix ID first obtains the ownership of the Matrix id. Since a smart contract can't make requests to external resources it's difficult to create a system which allows the smart contract to verify the actual ownership of the Matrix ID. Other systems do this by require a token from the user to confirm that they have access to the account. The server generates a token and it's sent directly to the user's account. But since a smart contract can't directly interact with the outside world a different process is needed.
A possible solution to this is to return a token when the user sets the Ethereum account address in the smart contract. The token then is communicated to the other communication parties who than can manually confirm the identity. This creates a mix of the manual and the automatic confirmation system. To additionally confirm the public list stored in the smart contract a function to confirm the identity by other users could be added.

Chapter 6

Reference Implementation

This section describes the reference implementation created to test the proposed design. The same software was also used to analyze the performance of the newly created system. The reference implementation was written in Rust to be coherent with the language used by OpenEthereum (the Ethereum client used) and Fractal (the Matrix client extended in this work).

The idea behind the design of the project was to have it work as a module

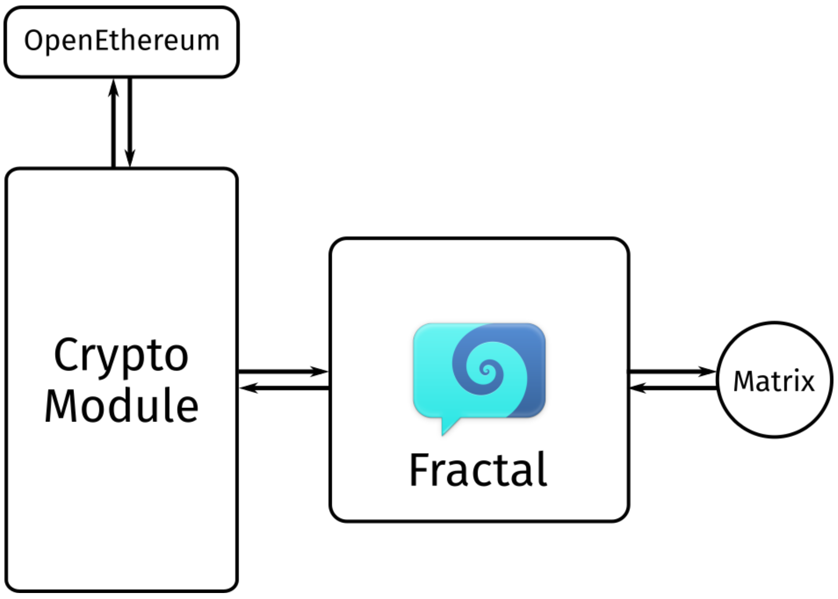


Figure 6.1: Overview of different modules in the reference application.

which can be integrated into other clients or even be used for end-to-end encryption of protocols other than Matrix. Therefore, the module is built as a library that handles all interaction with the Secret Store, including the Ethereum network and the smart contracts involved in the process of encryption and decryption of messages. The library also exposes a simple Application Protocol Interface (API) which allows a user of the library to encrypt and decrypt messages by calling only a few functions. This way the module can be used to integrate end-to-end encryption based on the Secret Store technology into any other application. Because it is written in Rust, the library even exposes a C API and can be easily integrated into applications written in other programming languages. The created library is published under the GNU General Public License (GPL-3.0-or-later) and the source code can be found at https://github.com/jsparber/e2e-secretstore.

6.1 Technical background

This section describes some of the technologies used to create the reference implementation.

6.1.1 Rust

Rust is a modern programming language mainly developed by Mozilla. Rust is a fast and memory efficient language because it does not require any runtime, and does not use a garbage collector. It was built to power performance-critical services and to be easily integrated with existing codebases in other languages[43]. With its small overhead it is perfect for embedded devices. The type system and ownership model built into Rust offers memory-safety and thready-safety guarantees which significantly reduces the number of typical bugs related to memory management and multithreading. The language also comes with well written documentation and the compiler has a user friendly interface with useful error messages and suggestions on resolving errors.

Rust has support for futures, which is a method to perform non-blocking asynchronous function calls. A future is basically a data structure that does not immediately return a result but a sort of pointer to the result. which will be yielded at some later point once the asynchronous task is completed. The reference implementation makes heavy use of this feature through the library tokio[1].

6.1.2 JavaScript Object Notation

JavaScript Object Notation, or much better known as JSON, is a lightweight data-interchange format[44]. JSON is based on the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999[44]. Even though it has JavaScript in its name the data structure is fully language independent. However, it contains many conventions from programming languages of the C-family (e.g. C, C++). The format is text based with a simple structure, which allows it to be easily readable, and manipulated and created by humans. By design, it's also easy to be parsed and generated by computers. For its nature JSON is an ideal format for transferring data between different programming languages and computer systems.
JSON is built on two different virtual structures. A list of key-value pairs called Object, and an ordered list of values called an Array. These structures are common in most modern programming languages, but they often have different names. Thanks to this it's simple to implement support for JSON in many different languages.
An example of a JSON is the following snippet, it contains an object with a key ”outer key” and a second key ”array”. The first key has another object as value, which itself has two key-value pairs. The key ”array” has an array as its value, with the numbers from 1 to 10.

```
{
  "outer key": {
```

[1]Tokio: https://tokio.rs/

```
    "inner key": "A random string",
    "data": "Some data"
  }
  "array": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
}
```

6.1.3 JSON-RPC

A remote procedure call (RPC) in distributed computing is used to execute a procedure, in most cases located on a remote computer, though this does not necessarily have to be the case. A remote procedure call is normally implemented so that it can be used in a similar way as local methods or functions. It is also possible to call a procedure on the same physical machine, therefore it can be used for inter-process communication. RPC abstracts many implementation details, like encoding and transport[45].
JSON-RPC is a remote procedure call protocol[46]. It is stateless and lightweight, and uses JSON as encoding. Like other RPC protocols JSON-RPC can have a wide range of transport protocols. In OpenEthereum it uses HTTP as a transport protocol[46].
Web3-rs is a library which allows to directly make RPC calls from Rust code. There are also other implementations of web3 for other languages to simplify the interaction with an Ethereum node.

6.1.4 Fractal

Fractal[2] is an Open Source Matrix client written in Rust. It's built by using the GTK graphical interface toolkit and it is part of the GNOME[3] project. The client is one of the few native applications a user can choose from to interact with Matrix. Fractal has a big audience of users and is actively improved and worked on. Like many other GNOME Projects, Fractal

[2]Fractal: https://gitlab.gnome.org/GNOME/fractal/
[3]GNOME: https://gnome.org

was created with a big focus on usability and interface design. The client does not currently implement Matrix end-to-end encryption, primarily because it is difficult to implement.
Fractal is split into two major parts. An API backend part, which communicates with the Matrix server, and a part that contains the GUI and data handling. Because Fractal is open source software and because of its modular structure it is simple to add new features. This property was used to add end-to-end encryption to Fractal to test the created library, and to experiment with the proposed designs.

6.2 Crypto Module

The created module is designed to work as a library, which can be integrated into any other software without much knowledge about how the module internally works. The library is fundamentally an abstraction from the Secret Store feature integrated in OpenEthereum.
The library has three distinct interaction points with the OpenEthereum node as shown in figure 6.2.

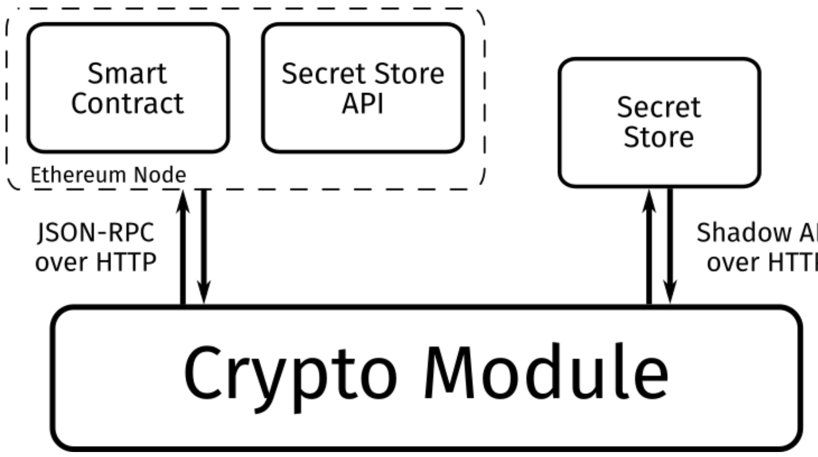


Figure 6.2: Overview of the internal structure of the reference implementation.

- Smart Contract. The library interacts with the Smart Contract, which is used as permissioning contract for the Secret Store. This interaction is done via the JSON-RPC interface of the OpenEthereum client.
- Secret Store API. This is the interaction point for operations performed on the users node. This includes encryption and decryption calls. All these calls are made via the JSON-RPC interface.
- Secret Store. This is the interaction point with the secret store. This interface is the only one which uses plain HTTP for communication.

6.2.1 Public API

The library exposes a simple interface to be used by client applications. The module has a single object to interact with. The object exposes three main methods:

- Encrypt. This method takes a document as input and encrypts it, and stores the encryption keys in the Secret Store.
- Allow access. This method adds the provided list of account addresses to the permissioning contract so that the specified address can gain access to the decryption keys.
- Decrypt. This method asks the secret store for the decryption keys. Only if the user is allowed to access the document they will receive the keys and decrypt the document.

6.2.2 Encryption

To encrypt a document the user needs to provide essentially three values to the crypto module:

- Document id. This can be any string. This reference implementation provides an auxiliary function that computes the sha256 of a provided document to standardize the used ID.

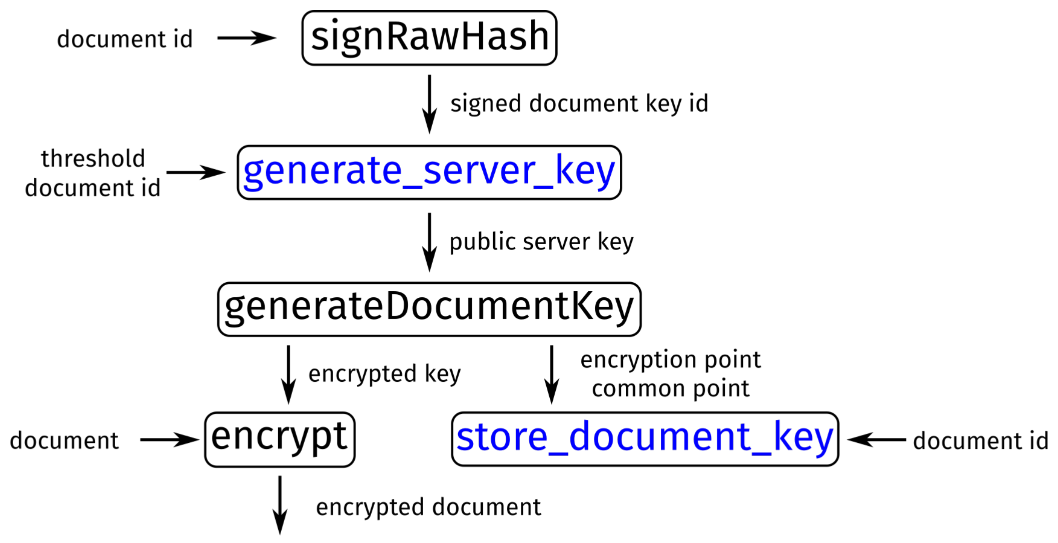


Figure 6.3: Flow of function calls to encrypt a document. The functions highlighted in blue are calls to the Secret Store node, the other calls are to the user's own node.

- Document. The message or data the user likes to encrypt.
- Threshold. The number of secret store nodes that need to agree on the permissioning contract state.

Figure 6.3 shows the process of encrypting a document. A document can be any data but it has to be encoded as hexadecimal string.
To encrypt a message the module internally talks to the user's node and to the Secret Store node. The interaction with the user's node is performed via JSON-RPC function calls. The Secret Store has a different interface which uses plain HTTP requests. The operations sent to the user's node and to the Secret Store in detail are:

- signRawHash. This computes recoverrable ECDSA (Elliptic Curve Digital Signature Algorithm) signatures. This method signs the 256-bit hash of the document id.
- generate_server_key (Secret Store). This method generates the server key, it is sent directly to the Secret Store node. This also specifics the

number of nodes needed to retrieve the private key, called threshold. This method returns the public portion of the generated key.

- generateDocumentKey. This method generates the document key in a fashion so that it remains unknown to all nodes of the Secret Store. It uses as input the public server key and returns all information needed to encrypt the document and to store the document key to the Secret Store.
- encrypt. This function encrypts the document.
- store_document_key (Secret Store). The final step is to store the document key to the Secret Store. So that it can be retrieved by a different user.

To allow other users to access the decryption key for a specific document identified via the document id the originator needs to give access to it by adding the address of allowed users to the permissioning contract. This is done by calling the dedicated function of the crypto module.

6.2.3 Decryption

Once a document was encrypted, the creator of the encrypted message needs to send the document ID and the encrypted document to the receiving peer.
When a user wants to decrypt an encrypted message they need to know the document ID and the encrypted document. The figure 6.4 shows the flow of information for the decryption. The specific steps performed by the crypto module to decrypt a message are:

1. signRawHash. This computes recoverrable ECDSA (Elliptic Curve Digital Signature Algorithm) signatures. This method signs the 256-bit hash of the document ID. This is the same method called as for encryption.

2. get_document_key (Secret Store). This method asks the Secret Store for the decryption keys. This request is sent directly to a Secret Store node. This only succeeds when the originator of the encrypted document has given the correct access permission to the receiver via the permissioning contract.

3. shadowDecrypt. This method uses the previously obtained keys to decrypt the encrypted document and returns the cleartext of the document.

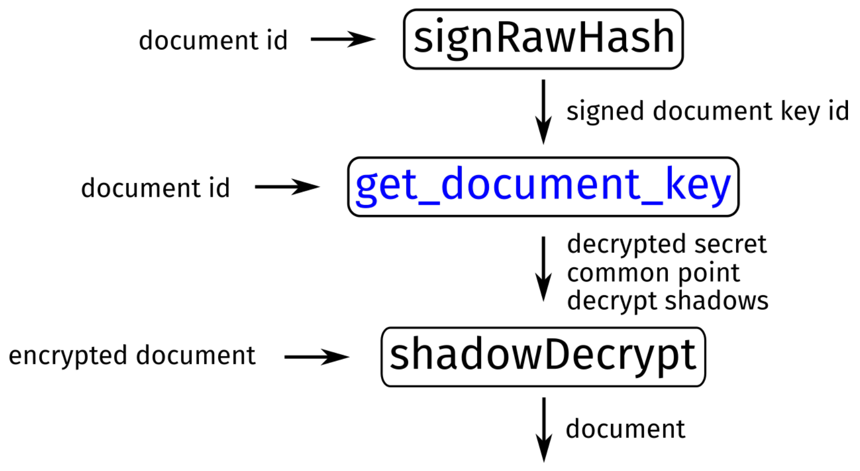


Figure 6.4: Flow of function calls to decrypt a previously encrypted document. The functions highlighted in blue are calls to the Secret Store node, the other calls are to the user's own node.

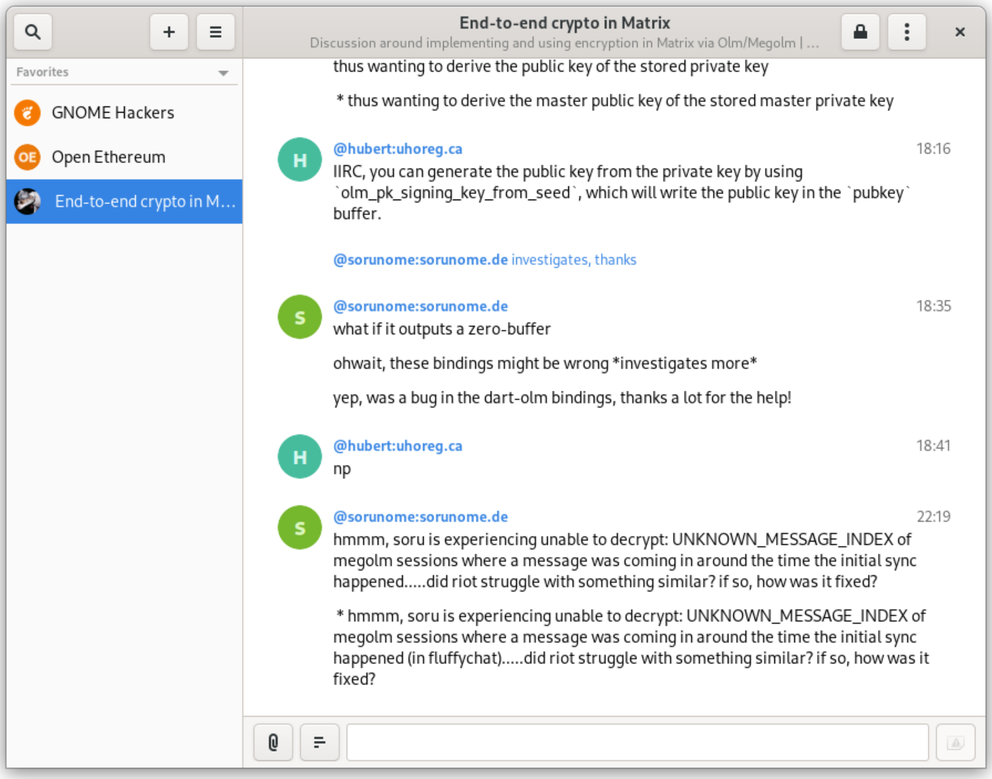
6.2.4 Permissioning contract

The crypto module also exposes two function to interact with the smart contract used as a permissioning contract for the Secret Store.

- allow_access. This function takes as input a document ID and a list of Ethereum account addresses. If the user has created this document and has stored keys for it in the Secret Store the permissioning contract will be updated and the address are included in the list of user allowed to decrypt the document.

- check_permission. This method calls the function of the smart contract that is used to check the permission to access the private keys stored in the Secret Store. This function's main purpose is to test the smart contract and to measure the time needed to execute the operation.

6.2.5 Client integration



Favorites
GNOME Hackers
Open Ethereum
End-to-end crypto in M…
End-to-end crypto in Matrix
Discussion around implementing and using encryption in Matrix via Olm/Megolm | …

thus wanting to derive the public key of the stored private key

* thus wanting to derive the master public key of the stored master private key

@hubert:uhoreg.ca 18:16
IIRC, you can generate the public key from the private key by using `olm_pk_signing_key_from_seed`, which will write the public key in the `pubkey` buffer.

@sorunome:sorunome.de investigates, thanks

@sorunome:sorunome.de 18:35
what if it outputs a zero-buffer

ohwait, these bindings might be wrong *investigates more*

yep, was a bug in the dart-olm bindings, thanks a lot for the help!

@hubert:uhoreg.ca 18:41
np

@sorunome:sorunome.de 22:19
hmmm, soru is experiencing unable to decrypt: UNKNOWN_MESSAGE_INDEX of megolm sessions where a message was coming in around the time the initial sync happened.....did riot struggle with something similar? if so, how was it fixed?

* hmmm, soru is experiencing unable to decrypt: UNKNOWN_MESSAGE_INDEX of megolm sessions where a message was coming in around the time the initial sync happened (in fluffychat).....did riot struggle with something similar? if so, how was it fixed?

Figure 6.5: Modified interface of Fractal. The button with the lock icon was added to allow users to enable end-to-end encryption.

The crypto module was built to be integrated into Fractal. So far Fractal does not have any kind of end-to-end encryption. However, its modular

design allows it to be easily extended.
Fractal's interface was modified to allow users to enable the end-to-end encryption for a specific room, as shown in figure 6.5. Since the end-to-end encryption method is built as an extension to the Matrix protocol, other clients that do not implement encryption via the Secret Store function normally but show an error message to the user that a message can't be decrypted, since of course they were not modified to support the new extension.

Chapter 7

Analysis and evaluation

This chapter contains the description of the performed tests to evaluate the proposed system. It also explains the expected results and compares them with the obtained results.
The setup for each test consists of a cluster of 25 computers, each running an instance of OpenEthereum. 24 nodes are running on the computers in the computer laboratory Ercolani of the University of Bologna. And one node is running on the computer executing the tests. The nodes at the University are connected with the local test computer over the Internet. The 25 nodes form the Secret Store and only these nodes obtain a share of the secret. Based on the set threshold a different number of nodes are involved in the encryption and decryption of a message.
In addition to the set of nodes forming the Secret Store, an additional node runs on the test computer that runs all user operations e.g. user interaction with the used smart contract. Figure 7.1 shows the complete test network setup. The created network is a private development chain that uses instant seal. This means that transactions are included into block that are instantly mined and verified.
To setup the network a custom shell script was used to generate the configuration for each node as well as to start and stop the different instances on all machines.

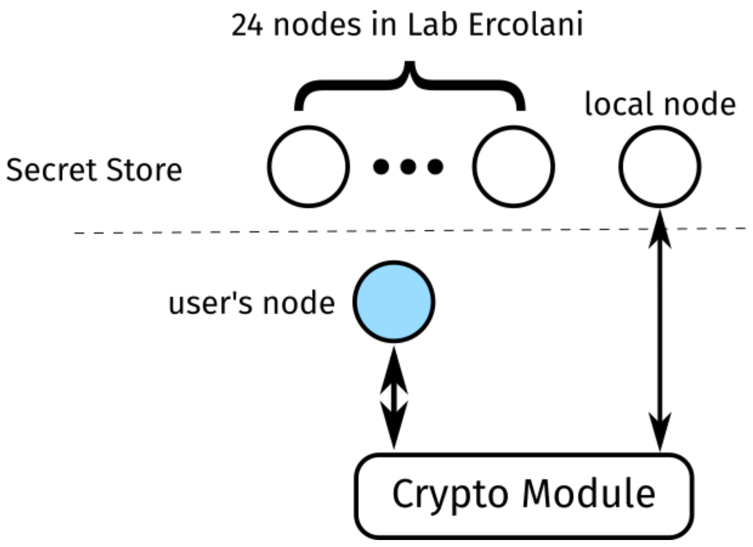


Figure 7.1: Test network setup. Each node is connected to all other nodes, hence the nodes form a fully connected Ethereum network. 25 nodes are part of the Secret Store. The node marked in blue is the only one not participating in the Secret Store, but it's home to the user's account.

In all the tests each single data point was obtained by running an operation five times and then the average value was calculated. This was done to retrieve more precise results.

The following tests were performed to evaluate the created system:

- Varying length of messages. Every parameter except the length of the encrypted message remains the same. The time of encryption and the time of decryption was measured.
- Varying threshold. This test keeps the message length fixed to 30 characters. The threshold was changed to measure the variation of time needed to operate.
- Access time to smart contract. This evaluation measures the access time to the smart contract. It varies the number of users allowed to decrypt a specific message.

- Gas usage. In this test the Gas usage is tracked. This evaluation varies the number users allowed to access a given message.

7.1 Varying length of messages

This test analyzes the time needed to encrypt and to decrypt of messages with different lengths. The length was varied from 100 to 4400 characters. The threshold was kept at a fix value of 1. The results shown in figure 7.2 are split into encryption time, decryption time and total time.

Since the Secret Store only stores the encryption keys that don't change

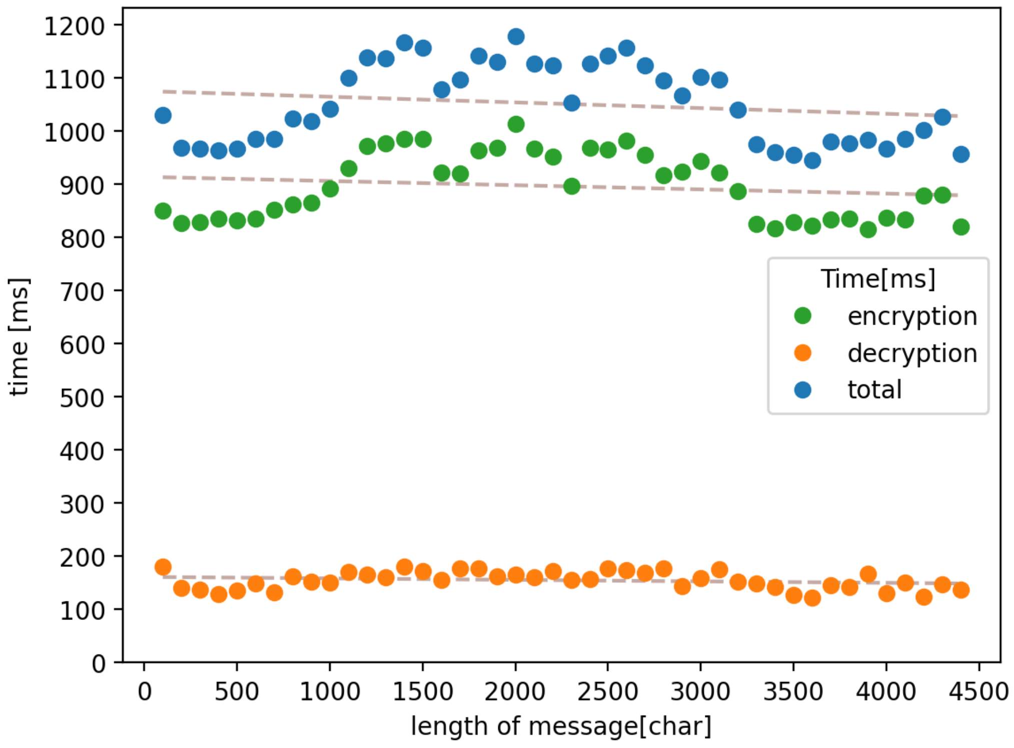


Figure 7.2: Time of encryption and decryption for messages with different length.

in size, not much variation was expected and therefore the result confirms the assumption. The small variation of operation time is most likely caused by variation of network speed, but overall the time needed doesn't depend on the size of the message and therefore the resulting curve is linear. Of course in a real public Ethereum network the times would be much higher since the time needed to confirm a transaction would be much higher and not instantly, like in this development chain.

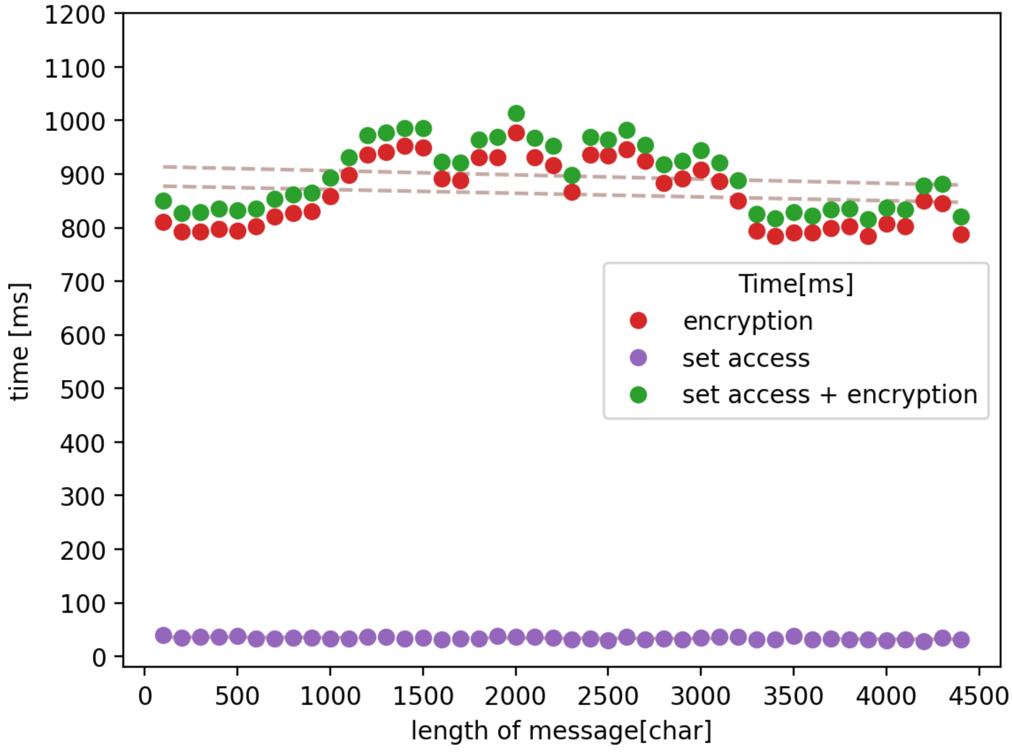


Figure 7.3: Time of encryption for messages with different length, split into encryption time and time to set the access permissions via smart contract.

Figure 7.3 shows the encryption time split into the time needed to generate the required keys and to encrypt the message, and the time required to set the correct access rights via the smart contract. As the previous graph 7.2 already showed, the time needed to encrypt a message doesn't depend on the message size.

7.2 Varying threshold

The threshold is a key parameter of the Secret Store. It defines how many nodes need to agree on the permissioning contract state and how many shares are required to access the decryption keys. Therefore this test tries different thresholds ranging from 1 to 24. Since the network only has 25 nodes the maximum possible threshold is 24, because the number of agreeing nodes needs to be bigger then the threshold. As figure 7.4 shows the encryption time remains mostly constant. The decryption on the other hand increases slightly with increasing threshold. This result was to expect because with a increasing threshold more and more nodes are involved in the key retrieval.

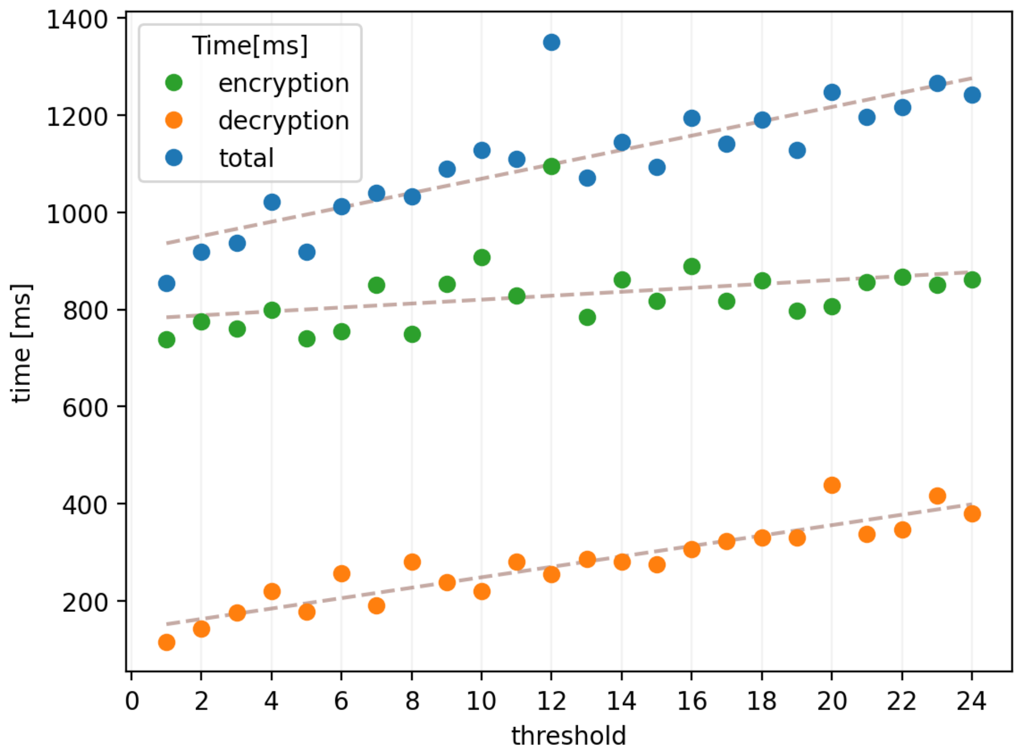


Figure 7.4: Time of encryption and decryption using different thresholds.

To observe the time to encrypt and to interact with the smart contract separately the results are reported in figure 7.5 with more detail. It's possible to see that the biggest time consumption comes from actually generating the keys and to encrypt a message. It's also visible that the curve is almost linear.

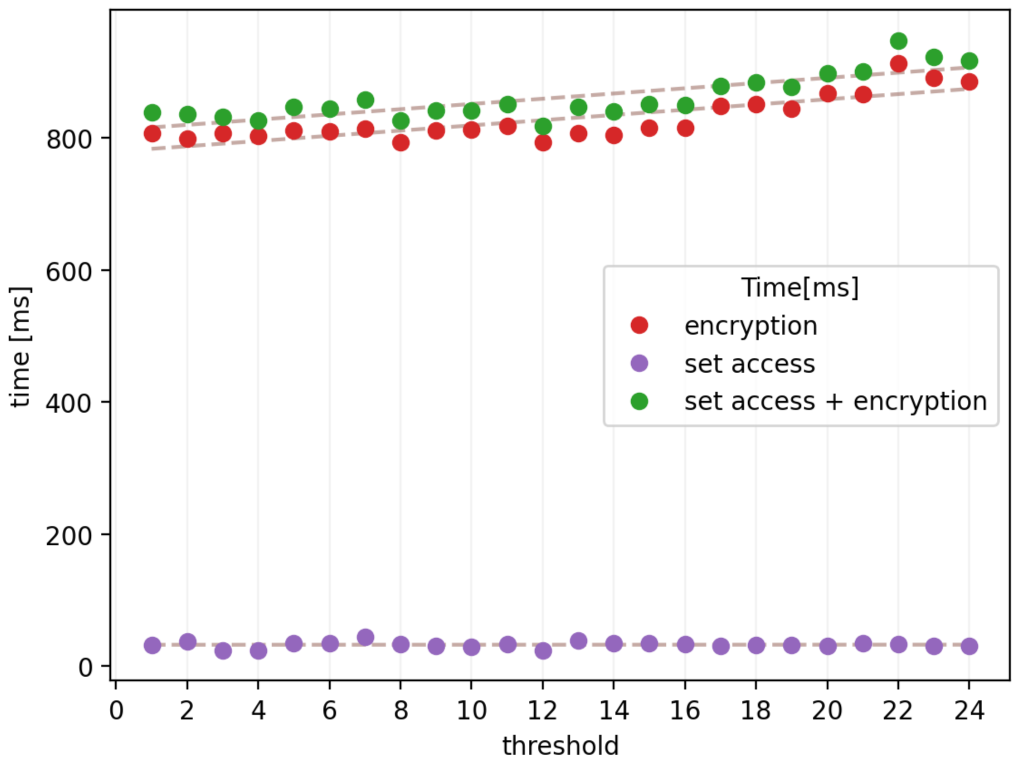


Figure 7.5: Time of encryption using different thresholds, split into encryption time and time to set the access permissions via smart contract.

7.3 Access time to smart contract

Another important measurement is the time required to set the users allowed to decrypt a message via the smart contract. Figure 7.6 shows the time required to set the users allowed to decrypt a message. It is called when encrypting a message, but could also be called at any later point to update the access permissions. The variable in this test is the number of users allowed to decrypt a message. It was varied from 10 to 360 users. The test results show that the time required is proportional to the number of users.

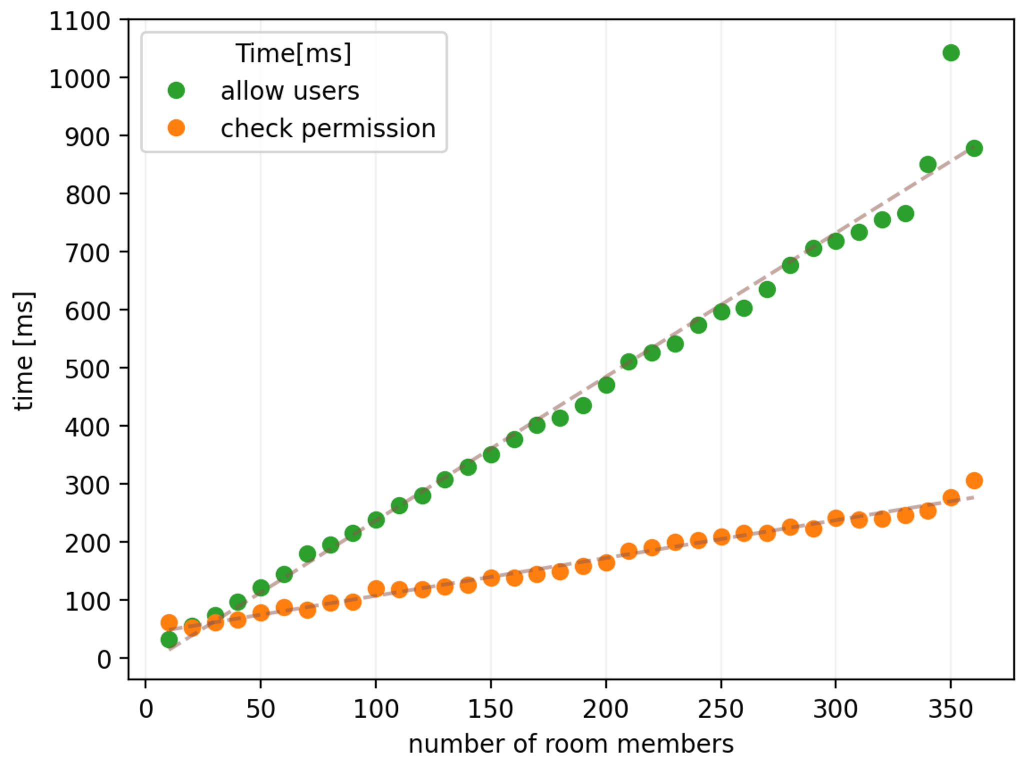


Figure 7.6: Time to set access permission with increasing number of allowed users, and the time to check permission via smart contract.

The second measurement in this test was the time needed to check the permission. This function is called when decrypting a message. It is called directly by each Secret Store node to verify if a user is allowed to obtain the decryption keys. This access time doesn't increase as much as the time needed to set the access permission but it increases noticeably with an increasing number of users.

7.4 Gas usage

Each call to a smart contract requires some amount of Gas, the crypto full used in Ethereum. Smart contracts have different types of functions. The functions that do not change the state (e.g. reading the memory) doesn't require any Gas. Functions that update the state or change the memory consume Gas which can be calculated by adding up the Gas consumption of each operation done by the function. This test evaluates the actual Gas used to set the access rights to allow the decrpytion of a message. Of course the system doesn't use any Gas for looking up the permission since it doesn't change the memory.

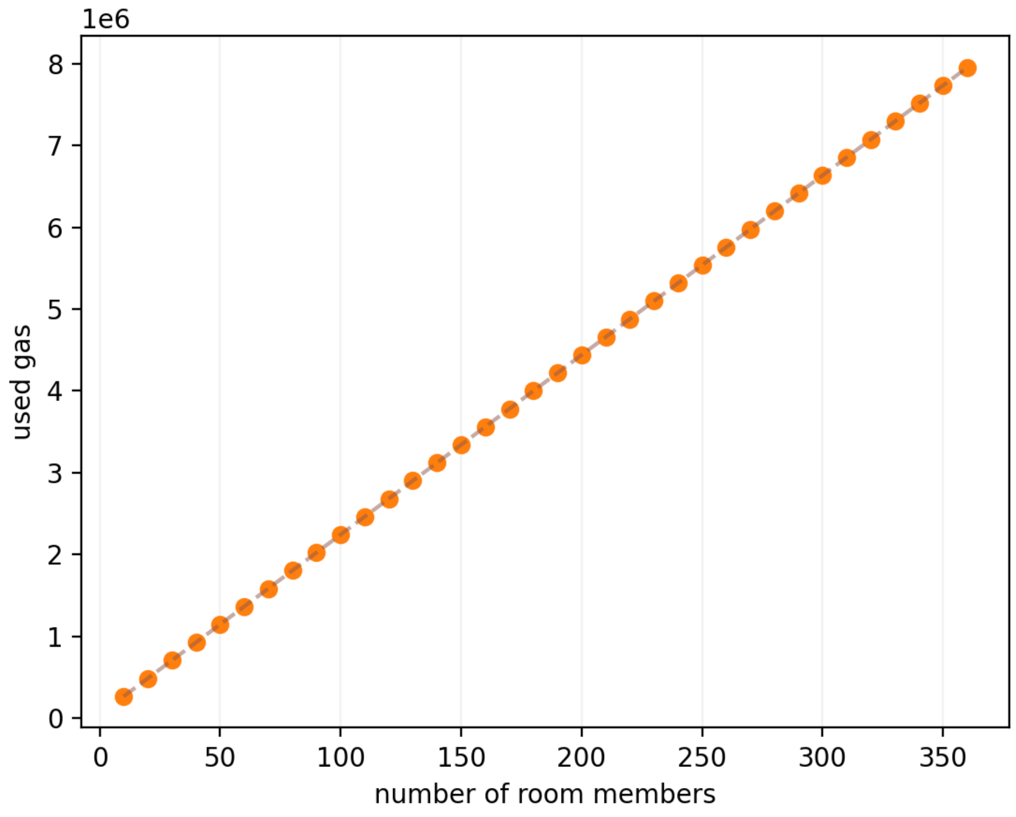


Figure 7.7: Gas used for setting access permission with increasing number of allowed users.

Figure 7.7 shows that the used Gas to set the access rights is proportional to the number of users allowed to access the message. This was to be expected because the *allow_access* function uses a for loop. Eliminating the iteration over the list of allowed users would lower the Gas consumption. It would also be of desire to make the Gas consumption independent from the number of specified addresses.

Conclusion

In this work a system for end-to-end encryption was proposed. It is built on top of the Secret Store feature integrated into OpenEthereum, so the security of the system is guaranteed via the blockchain-based Ethereum computing platform. The created scheme was used to implement a library that can be used to add end-to-end encryption for different communication protocols, with a special focus on the Matrix instant messaging protocol.
The proposed design was built with non-technical users in mind, so even people who don't know anything about the system can easily use it. The encryption of a communication involves two distinct processes. The process of encryption and decryption is straightforward and can be implemented without any user interaction, but the verification of a user is still an open issue much researched because it generally requires some user interaction to verify the identity of the other communication parity. In this work two different solutions for this issue were discussed, as well as a mix between the two possible solutions. The mixed solution is composed of a public database of identities and a manual verification. To store the public identities, it utilizes a smart contract.
The Ethereum blockchain is a secure way to store the data, therefore, the proposed system eliminates the need of backups of the keys and as long as the user maintains the correct access permission in the smart contract they can't ever lose access to send messages.
This work only includes a relatively simple permissioning contract, but it would be possible to create a much more advanced system for controlling the

access to a communication. Thanks to smart contracts it would be possible to create a system that allows to give permission to access a message to a new user joining the conversation at a later point or also to revoke the permission at some point in the future. Other end-to-end encryption systems leak this ability because they normally require to encrypt a message specifically for each user.

The reference implementation written to test the proposed design was created in the form of a library. The library exposes a simple application programming interface (API). Thanks to this library Fractal, the Matrix client modified in this work, has gained the ability to end-to-end encrypt messages. Even though the library was built for the integration into Fractal, it was designed to be completely agnostic of the transport protocol, hence the same library can be used without much trouble to add end-to-end encryption to other communication protocols.

Multiple tests were performed to evaluate the performance and to analyze the speed of the created system. Each test was executed on a network of 25 Ethereum nodes. To run the nodes the computers of the computer lab Ercolani at the university of Bologna were used.

The tests performed to evaluate the time needed to encrypt a message showed that the time required is independent from the message size as well as from the threshold. The time required to decrypt a message, on the other hand, increased when the threshold was increased, but just as with the encryption time the decryption time isn't connected to the size of the message. This is because only the key is stored in the Secret Store, not the message itself. Even though the threshold has an impact on the speed of the system, the time needed to encrypt a message stays around 1s and the time needed to decrypt a message stays below 400ms with a threshold of 24 nodes.

The test performance to evaluate the speed of the interaction with the smart contract was run with different numbers of users since this is the only variable. This test showed that the time needed to allow a list of users to access a message is proportional to the size of the list. The call to the function

check_permission also increases with the size of the set of allowed users but much more slowly than the function used to allow users to access a message. Also, the Gas consumption to execute the permissioning contract was analyzed. The function *check_permission* doesn't consume any Gas because it doesn't modify the memory, but the function *allow_access* requires a Gas amount proportional to the number of users specified in the function call.
The test results show that even with a threshold of 24, the time needed to encrypt a message is about 1s. However, running the same operation on a non-development network could significantly decrease the encryption speed because the transaction would not be executed instantly. To increase the speed, a combination of the Secret Store and a symmetric-key encryption scheme could be used, where the Secret Store is used to share a symmetric key. Generally, symmetric-key schemes are much faster than public-key encryption systems. This would significantly reduce the overhead needed for the encryption of each message.
The test results show that the proposed end-to-end encryption system is scalable for a large number of messages and for large message sizes, but a large number of users can slow down the interaction with the smart contract. This issue can be solved by collecting the users of a room into a virtual group, which then allows to give permission to access a message in bulk.

Bibliography

[1] The Matrix.org Foundation CIC. Matrix [homepage]. https://matrix.org/, 2019. [Online; accessed 20-April-2020].

[2] Hubert Chathi. An introduction to end-to-end encryption in Matrix and Riot. https://www.uhoreg.ca/blog/20170910-2110, 2017. [Online; accessed 11-April-2020].

[3] Parity ethereum. https://openethereum.github.io/wiki/Parity-Ethereum. [Online; accessed 21-May-2020].

[4] Secret Store. https://openethereum.github.io/wiki/Secret-Store. [Online; accessed 21-May-2020].

[5] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition.* Chapman Hall/CRC, 2nd edition, 2014.

[6] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners.* 01 2010.

[7] John Wagnon. Real Cryptography Has Curves: Making The Case For ECC. https://devcentral.f5.com/s/articles/real-cryptography-has-curves-making-the-case-for-ecc-20832. [Online; accessed 19-June-2020].

[8] Giorgio Zanin. Secret Sharing Schemes and their Applications. http://wwwusers.di.uniroma1.it/smart/ppt/zanin.pdf. [Online; accessed 24-April-2020].

[9] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[10] *Foundations of Applied Mathematics, Volume 2 : Algorithms, Approximation, Optimization.* SIAM, 2020.

[11] Alfred V. Aho, Jeffrey D. Ullman, and John E. Hopcroft. *The *design and analysis of computer algorithms.* Addison-Wesley, 1974.

[12] Andy Greenberg. Hacker Lexicon: What Is End-to-End Encryption? `https://www.wired.com/2014/11/hacker-lexicon-end-to-end-encryption/`, 2019. [Online; accessed 24-April-2020].

[13] Moxie Marlinspike Trevor Perrin. The Double Ratchet Algorithm. `https://signal.org/docs/specifications/doubleratchet`, 2016. [Online; accessed 7-May-2020].

[14] Moxie Marlinspike. WhatsApp's Signal Protocol integration is now complete. `https://signal.org/blog/whatsapp-complete/`, 2016. [Online; accessed 7-May-2020].

[15] Wire Swiss GmbH. Wire Security Whitepaper. `https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf`, 2018. [Online; accessed 7-May-2020].

[16] The Matrix.org Foundation CIC. End-to-End Encryption implementation guide. `https://matrix.org/docs/guides/end-to-end-encryption-implementation-guide`, 2019. [Online; accessed 10-April-2020].

[17] Moxie Marlinspike. The Double Ratchet Algorithm. `https://signal.org/blog/advanced-ratcheting/`, 2013. [Online; accessed 7-May-2020].

[18] Pierre Karpman Ange Albertini Yarik Markov Marc Stevens, Elie Bursztein. The first collision for full sha-1. 2017.

[19] ConsenSys. Are you really using SHA-3 or old code? `https://medium.com/@ConsenSys/are-you-really-using-sha-3-or-old-code-c5df31ad2b0`, 2016. [Online; accessed 29-April-2020].

[20] The Economist. Who is Satoshi Nakamoto? `https://www.economist.com/the-economist-explains/2015/11/02/who-is-satoshi-nakamoto`, 2015. [Online; accessed 13-May-2020].

[21] Imran Bashir. *Mastering Blockchain.* Packt Publishing, 2017.

[22] Maarten : van Steen and Andrew S. Tanenbaum. *Distributed Systems.* CreateSpace Independent Publishing Platform, 2017.

[23] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[24] Georg Becker and Ruhr-Universität Bochum. Merkle signature schemes, merkle trees and their cryptanalysis, 2008.

[25] Azaghal. Image of hash tree. `https://commons.wikimedia.org/w/index.php?curid=18157888`, 2012. [Online; accessed 7-May-2020].

[26] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. 4(3), 1982.

[27] Ameer Rosic. Proof of work vs proof of stake: Basic mining guide. `https://blockgeeks.com/guides/proof-of-work-vs-proof-of-stake/`, 2017. [Online; accessed 7-May-2020].

[28] Bitcoin energy consumption index. `https://digiconomist.net/bitcoin-energy-consumption`. [Online; accessed 7-May-2020].

[29] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. 2012.

[30] Delegated proof-of-stake consensus. https://bitshares.org/technology/delegated-proof-of-stake-consensus/. [Online; accessed 7-May-2020].

[31] Nichanan Kesonpat. Dpos. https://www.nichanank.com/blog/2018/6/4/consensus-algorithms-pos-dpos, 2018. [Online; accessed 7-May-2020].

[32] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 164–186, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[33] Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security*, pages 494–509, Cham, 2017. Springer International Publishing.

[34] Openzeppelin. https://openzeppelin.com, 2017-2019. [Online; accessed 25-June-2020].

[35] Overview of dapp development. https://openethereum.github.io/wiki/Deploying-Dapps-to-Parity-UI.html. [Online; accessed 21-May-2020].

[36] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In Rainer Böhme and Tatsuaki Okamoto, editors, *Financial Cryptography and Data Security*, pages 507–527, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[37] web3.js. https://github.com/ethereum/web3.js/blob/0.15.0/lib/utils/utils.js#L40. [Online; accessed 21-May-2020].

[38] Solidity. `https://solidity.readthedocs.io/en/v0.6.8/`, 2016-2020. [Online; accessed 24-May-2020].

[39] The Matrix.org Foundation CIC. Introduction [to Matrix]. `https://matrix.org/docs/guides/introduction`, 2019. [Online; accessed 20-April-2020].

[40] Ben Parsons. Dweb: Decentralised, Real-Time, Interoperable Communication with Matrix. `https://hacks.mozilla.org/2018/10/dweb-decentralised-real-time-interoperable-communication-with-matrix/`, 2018. [Online; accessed 20-April-2020].

[41] The Matrix.org Foundation CIC. Matrix Specification. `https://matrix.org/docs/spec/`, 2014-2019. [Online; accessed 20-April-2020].

[42] Megolm group ratchet. `https://gitlab.matrix.org/matrix-org/olm/-/blob/master/docs/megolm.md`. [Online; accessed 12-April-2020].

[43] Rust. `https://www.rust-lang.org/`. [Online; accessed 24-May-2020].

[44] Introducing JSON. `https://www.json.org/json-en.html`. [Online; accessed 15-April-2020].

[45] Dave Marshall. Remote Procedure Calls (RPC). `https://users.cs.cf.ac.uk/Dave.Marshall/C/node33.html`, 1999. [Online; accessed 20-April-2020].

[46] JSON RPC API. `https://wiki.parity.io/JSONRPC`. [Online; accessed 20-April-2020].