Alma Mater Studiorum · Università di Bologna

Campus di Cesena
Scuola di Ingegneria e Architettura

Corso di Laurea Magistrale in
Ingegneria e Scienze Informatiche

Ethereum blockchain as a decentralized and autonomous key server: storing and extracting public keys through smart contracts

Tesi Magistrale in Sicurezza delle Reti

Relatore:
GABRIELE D'ANGELO

Presentata da:
PIER FRANCESCO COSTA

II Sessione
Anno Accademico 2016/2017

Dedicated to my family,
for their love and support
throughout my life.

Abstract

Ethereum is an open-source, public, blockchain-based distributed computing platform featuring smart contract functionality. It provides a decentralized Turing-complete virtual machine which can execute scripts using an international network of public nodes. The purpose of this thesis is to build a decentralized and autonomous key server using Ethereum smart contracts to store and retrieve information. We did an overall introduction of Bitcoin and Ethereum to provide a background of the study. We then analyzed the current problems of key discovery with traditional servers and web-of-trust. We designed, built and tested an application that can verify contact cards (email address, PGP public key, domain address, Facebook account), link them to an Ethereum address and store them on a public contract running on the Ethereum blockchain. Finally we made an analysis of the costs and limitations of such solution and proposed some future improvements. The results show that Ethereum is a good choice for storing public keys, thanks to the immutability and irreversibility of the blockchain.

Contents

Introduction

In our society the quantity of information exchanged electronically over the Internet has sustained rapid growth over the past years, and it is predicted to grow even more sharply. Important communications, like business deals, financial transactions, medical records, private material, etc. need to be relied over secure and authenticated channel.

After realizing the pervasiveness of Internet surveillance conducted both by nation state actors and criminal organizations, many citizen and companies have concluded that the only way to achieve privacy and secure communication over the Internet is using end-to-end encryption. PGP, one of the most widely used email encryption standard, enables users to send secure emails over insecure channels without fear of interception or alteration of the content.

This standard, like many others security mechanism based on public key encryption, has one important shortcoming: key discovery. Before establishing a secure channel, the sender must in fact first retrieve the public key of the recipient. But doing that over an insecure channel is obviously not recommended, because an attacker could easily change the authentic public key with his own, enabling a man in the middle attack.

Various techniques has been adopted for securely establishing a link between a public key and its owner identity, like hand exchange, web of trust and public key infrastructure.

In this thesis we will propose a new method for secure key discovery. The solution will involve designing and building an autonomous and decentralized key server based on Ethereum smart contracts. This system will enable users to store their PGP public keys and other contact information on the Ethereum blockchain, instead of relying on a traditional key server. This will reduce the amount of trust that users have to put on third party services for their secure communication needs. Having to trust different entities is always a liability under the security point of view, because increasing the number of actors leads to an increased probability that one of them will be compromised by an attacker or act maliciously.

In the first chapter we will explore some background concepts necessary to understand the thesis argument. First a brief history of blockchain as a concept, then some essential cryptography concepts. After this we will try to explain briefly the working how the two more popular cryptocurrencies: Bitcoin and Ethereum. We will then evaluate advantages and disadvantages of both technologies.

Over the last few years blockchain distributed systems have been growing both in term of capabilities and in term of popularity among many industries, especially the financial and banking industry. More and more insiders are praising blockchain as a revolutionary tool that will reshape the landscape of software and Internet technologies. Lot of so called "blockchain startups" have sprung all over the world claiming that they can improve every aspect of a distributed system, such as reliability, fault tolerance, security, decentralization and extensibility just by putting a blockchain somewhere in the system architecture. It is therefore very difficult to understand the real implications of this technology through all the noise and speculations.

Bitcoin represents the first and most successful example of blockchain based cryptocurrency. On the other hand, Ethereum is the bleeding edge of the innovation in the sector and introduces a lot of interesting concepts, like explicit modeling of smart contracts and Turing-complete computation capabilities.

The second chapter will focus on two traditional solutions for PGP key discovery, web of trust and key servers, and their shortcomings, and then will explain our proposal. There will be an evaluation of different strategies to verify user's information data, with their respective advantages and disadvantages.

In the third chapter we will lie down the requirements for the proposed solution and, after a phase of problem analysis, we will build a logical model under three dimensions: interaction, structure and behaviour.

The fourth chapter will show the tools used to build the application, the reason behind such choices and the practical implementation of the model. This part will include example of the code used in the application and screenshots of the graphical interface.

In the fifth and final chapter we will present the results of our work and draw the conclusions. We will analyze advantages, disadvantages and costs of our implemented solution and the differences between the model and the actual implementation. We will also discuss future work and prospects for the application.

Chapter 1

Background

1.1 Blockchain history

The concept of decentralized digital currency has been around for decades. It started to grow mainly in the cypherpunk environment, which has the vision to achieve social and political change through the use of cryptography [1]. A form of electronic cash without an issuer or any form of middle man can allow an unprecedented degree of security and privacy compared to the traditional banking system. The first prototypes of electronic cash, developed during the 80s and 90s used David Chaum's blind signatures to achieve privacy and authentication [2, 3]. The main limitation of such approach was that a trusted entity was still required to issue the currency. The absence of this entity would mean that each user could generate an arbitrary amount of money, making the system useless.

The next step was trying to decentralize the issuance of the currency. One of the solution proposed allow users to create new currency after showing proofs that they have solved some hard mathematical problem. So each user could issue an amount of currency proportional to their computing power. This way the amount of money generated could tied to some physical good, like the electricity and the hardware required to solve the problem. These ideas where first exposed by Wei Dai in his paper *B-money* [4] in 1998 and later expanded by Hal Finney, using Adam Back's Hashcash [5] algorithm as a form of reusable proof-of-work [6].

The blockchain has been introduced for the first time with Bitcoin project. The concept is described by the white paper *Bitcoin: A Peer-to-Peer Electronic Cash System* [7] released during 2008 by an anonymous author under the pseudonym of Satoshi Nakamoto. The aim of the author was realizing a peer-to-peer electronic cash system without a single authority issuing the

currency. The most innovative thing in Bitcoin is the consensus mechanism and the issuance of the currency, based both on a proof-of-work competition between users.

The original Bitcoin open source client (version 0.1) was released to the public on the 9th of January 2009, while the first block of the blockchain, called genesis block, was generated six days before. For the first year the network growth was slow and predictable, the adoption was driven mostly by developers and libertarian idealists that considered it just like an interesting test or curious proof-of-concept, without any concrete economic value. The first currency exchange, Mt.Gox (now in liquidation) started operating in 2010, giving, for the first time, to the Bitcoin currency (called in this work with the acronym BTC) a value in term of fiat money[1].

During the following years the value of the currency grew rapidly, driven both by adoption as a mean of exchange for goods and services over the Internet, speculation and media exposure. Starting from few cents in 2010 the price of 1 BTC reach the parity with the US Dollar on February 2010, and in just few years it peaked at around 1000 $ on 2013, and after few years reached a price of over over 4000 $ in August 2017 [9].

Figure 1.1: Bitcoin price chart in US Dollar

The success of Bitcoin as a currency has been undeniable, even if it didn't reach the mainstream status as a new global standard some of it proponents

[1]Money without intrinsic value that is used as money because of government decree [8].

have been hoping. Digital money has historically been implemented by bank and financial institution, usually as credit card transactions or wire transfers. The traditional way of moving money online involves heavy regulated institutions that the user have to trust, and for this reason opening accounts is long, costly and time consuming in the developed world, while is sometimes not possible for large portions of potential users living in developing countries. It is not a mystery that lot of friction is still present between users' demands and banks' offers in the traditional banking industries.

Bitcoin can be identified as a solution for a majority of those problems. Some of it advantages over traditional payment systems are:

- low transaction cost (at least until 2016[2]),
- no need to rely on a central authority with the ability to block or revert movements,
- no identities requirements for opening accounts,
- no risk of issuer bankruptcy,
- privacy through default pseudonymity and optional anonymity.

Despite these evident merits, there are some disadvantages that have stopped Bitcoin from becoming a popular payment app like, for example, PayPal: the fact that the value of a BTC is variable in relation to traditional currencies like Euro and Dollar, the relative difficulty of buying BTCs for the typical user and a the perception that they are insecure or used by criminals [12].

1.2 Cryptography

To understand the fundamentals concepts of Bitcoin, cryptocurrencies and blockchains in general, a basic understanding of computer cryptography is required. This section obviously will be just a remainder of the important

[2]Bitcoin has currently a limit on the size of each block set at 1 MB, that caps the number of transaction that can process at every given time. In 2017 this limit has been reached, so user started increasing the fee included in the transaction as a way to convince miners to include their transaction in a block as soon as possible. This led to an increase of the average transaction fee paid on the network, from around 0.20 $ in 2016, to 2-6 $ in 2017 [10]. Moreover, the Bitcoin development community, locked in a bitter discussion around this problem, called scaling issue, has not yet reached a consensus to solve it, like removing the 1 MB limit or adopting other solutions [11].

concepts, so it is suggested, if the reader has no knowledge of the subject, to consult some material about cryptography, like Bruce Schneier's *Applied cryptography: protocols, algorithms, and source code in C* [13].

Cryptography is the practice and study of techniques for secure communication in the presence of third parties called adversaries [14]. It is used, in information technology, to achieve confidentiality, integrity, authentication, and non-repudiation.

Bitcoin, and every other blockchain system, relies heavily on digital signatures. A digital signature algorithm, when applied to a message, can achieve the following goals:

Authentication The receiver of a message can trust the sender identity;

Integrity The receiver of a message can trust that the content of the message have not been tampered;

Non-repudiation The sender of a message can not later deny that message was sent.

The purpose of a digital signature can be compared to that of a traditional hand-written signature, but properly implemented digital signature are much harder to forge.

Digital signature algorithms usually exploit two additional cryptographic techniques: **hashing algorithms** and **public key cryptography**.

1.2.1 Hash function

A hash function is a mathematical function that accepts an arbitrary size input and return a fixed size output. The value returned by a hash function is called hashes or hash sum 1.7. This work will only consider a very specific sub class of hash functions, called cryptographic hash functions [13].

A cryptographic hash function is designed to be a one-way function, or in other words a function that is really difficult to invert. Because of that, the fastest way to find the input of and ideal cryptographic hash function's output is to try all the possible inputs until the output matches (brute force search). Another important property of a cryptographic hash function is that that a small change in the input should produce a completely different and seemingly random output (avalanche effect).

Cryptographic hash functions are fundamental in many areas of computer security, such as digital signatures, message authentication codes, fingerprinting, detecting accidental corruption. They can also be used to index data in hash tables.

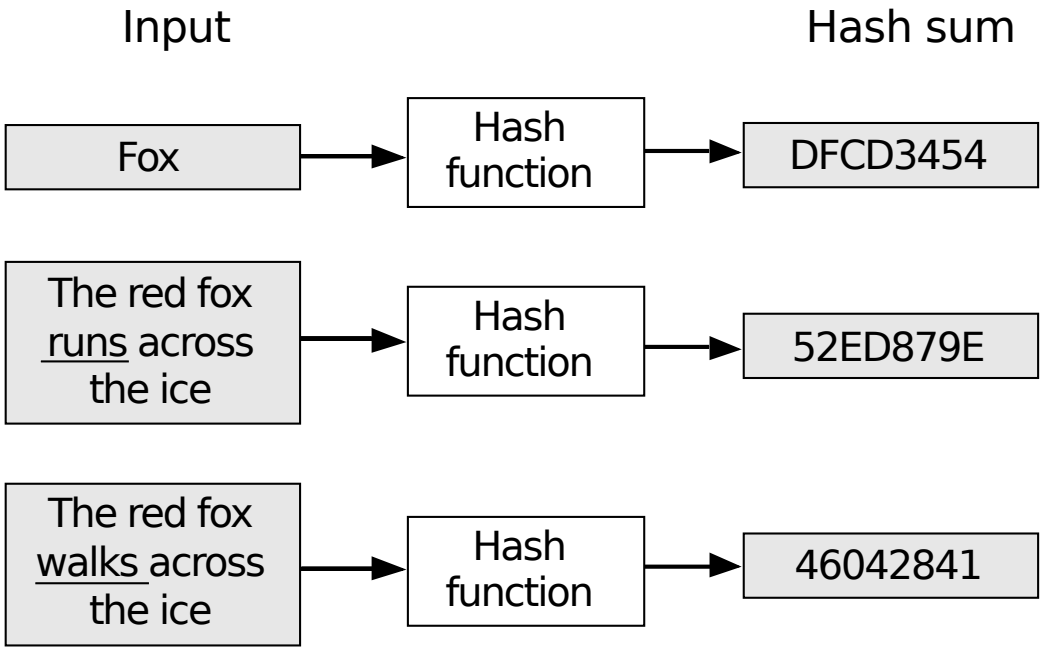

Figure 1.2: Hash function. Even similar inputs generate completely different outputs [15]

These functions play a crucial role in any blockchain implementation and are commonly used both for fingerprinting and as proof-of-work algorithms.

1.2.2 Public key cryptography

Symmetric cryptography, even before the invention of computers, has been used by humans for millenia as a way to hide important information from adversaries, mostly in military communication.

An algorithm that implements it has two inputs, plain text and a secret key, and one output, the encrypted data. The only piece of information that must remain secret to a malicious attacker is the secret key, while the algorithm, if well designed, can be made public. Decryption of the data is usually done in reverse. The secret key and encrypted data act as inputs of the decryption algorithm (depending on the actual algorithm, it can be the same used for encryption), while the plain text data is the output. It is called symmetric because the same key is used for both encryption and decryption[Figure 1.3].

Using symmetric cryptography to send secure messages is somewhat impractical, because both the sender and the receiver must know in advance

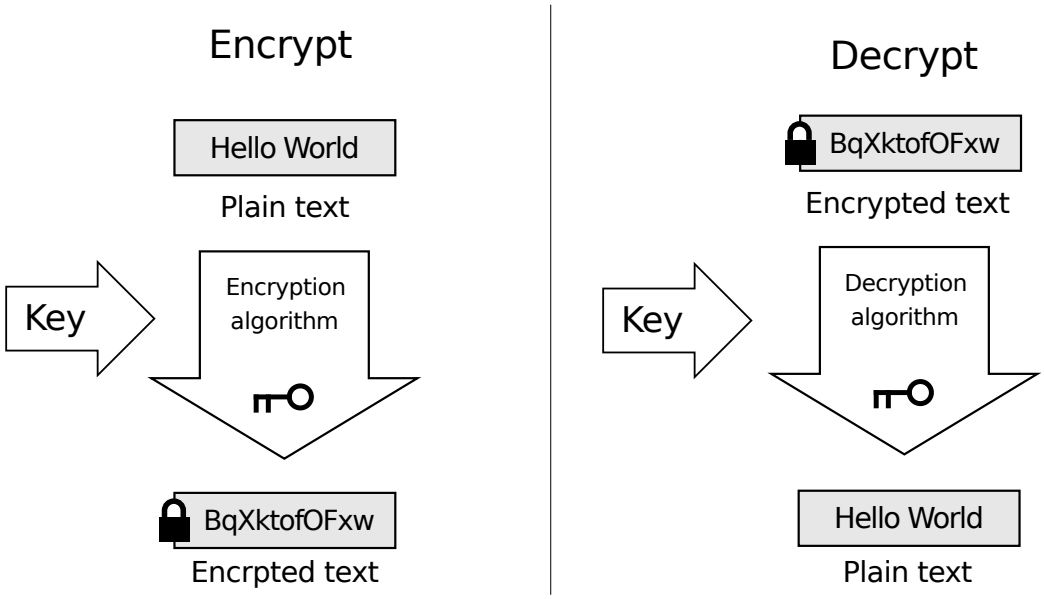

Figure 1.3: Symmetric encryption [16]

the secret key used to encrypt and then decrypt the message. So key must be exchanged through a secure channel of communication before the encrypted message is sent. But a secure channel can be difficult to obtain or impractical, and if not so, then it can be used as the main communication channel without using cryptography at all.

Public key cryptography, also called asymmetric cryptography, solves this problem in an elegant way. Two different keys are used with the same encryption algorithm. If the data is encrypted with one key, it can be decrypted only using the other key, and vice versa. One key is referred as private key, while the other as public key, depending on their role in the scheme. The private key is randomly generated, while the public key is derived from the private key through a one way function. In this way is not possible to derive the private key knowing just the public key.

With public key cryptography is possible to implement both encrypted communication and digital signature mechanisms.

Encrypted communication The recipient of the message shares publicly his public key. The sender encrypts the message with the recipient's public key. The encrypted message is sent and only the recipient, owning the private key, can decrypt it.

Digital signature The sender of the message shares publicly his public key.

Then he encrypts the hash of the message with his private key and sends both the plain text message and the encrypted hash to the recipient. [Left part of figure 1.5].

The receiver then proceeds to decrypt the hash with the public key of the sender. Finally he computes the hash of the received message and compare it with the decrypted hash. If the hashes match the message was indeed signed by the sender. [Right part of figure 1.5]

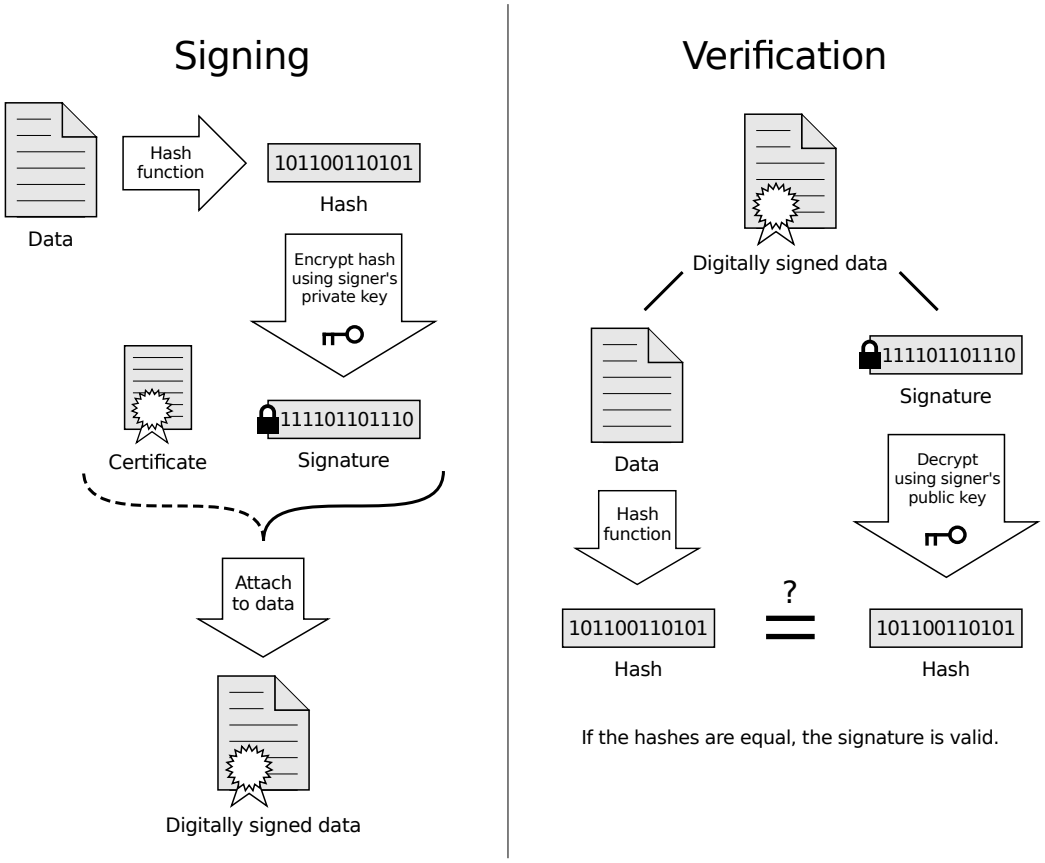

Figure 1.4: Public key digital signature [16]

Using these two mechanism together is possible to achieve data confidentiality, data integrity, authentication, and non-repudiation.

1.2.3 Pretty Good Privacy

Pretty Good Privacy (PGP) is an encryption program created by Phil Zimmermann in 1991 to encrypt, decrypt and sign texts, emails, files, directories

and disk partitions [17]. It follows the OpenPGP standard (RFC 4880), and is just one of many different implementations (OpenPGP.js, GPG, etc.).

A user, just knowing the PGP public key of the recipient can use it to encrypt and sign a message, and then send it over unencrypted mail. The recipient will be able to verify the authenticity sender thanks to the signature and will be the only able to read it thanks to encryption. PGP can also be used to just encrypt something without signing it, or signing something without encrypting it.

PGP, being an hybrid cryptosystem, combines some of the best features of both symmetric and public key cryptography. To encrypt plaintext, first some compression is used to reduce the seize of it and also to make more difficult do discover something about the nature of the text once encrypted through cryptanalysis[3].

PGP then generates a random session key. This key is used to encrypt the compressed plain text using a symmetric encryption algorithm. This is done because conventional encryption is much faster than public key encryption on large text.

Then the session key is encrypted with the recipient's public key using an asymmetric algorithm. This public key-encrypted session key is transmitted along with the ciphertext to the recipient. If needed the sender can also sign the message: the hash of the plaintext is signed with the sender public key, and it is attached to the message before compression.

To read the message the recipient has to decrypt the session key with his private key. Then he can use the session key to decrypt the message and decompress it. If signed the recipient can also check the signature validity. He must decrypt the signed hash using the sender's public key, compute the hash of the message independently and compare them: if they are equal that means that the signature is valid and that the message ha not been altered.

The OpenPGP standard define which algorithms different PGP implementation can support for different purposes (mandatory in bold) :

- **Message encryption: CAST5**, IDEA, **3DES**, Blowfish and **AES**
- **Session key encryption: RSA** and **Elgamal**.
- **Signature: DSA** and **RSA**.
- **Hashing: SHA-1**, RIPE-MD/160, SHA256 and others.
- **Compression: ZIP**, ZLIB and BZip2.

[3]Cryptanalysis is the science of analyzing and breaking secure communication. Some methods involve searching for patterns or regularities inside encrypted text.

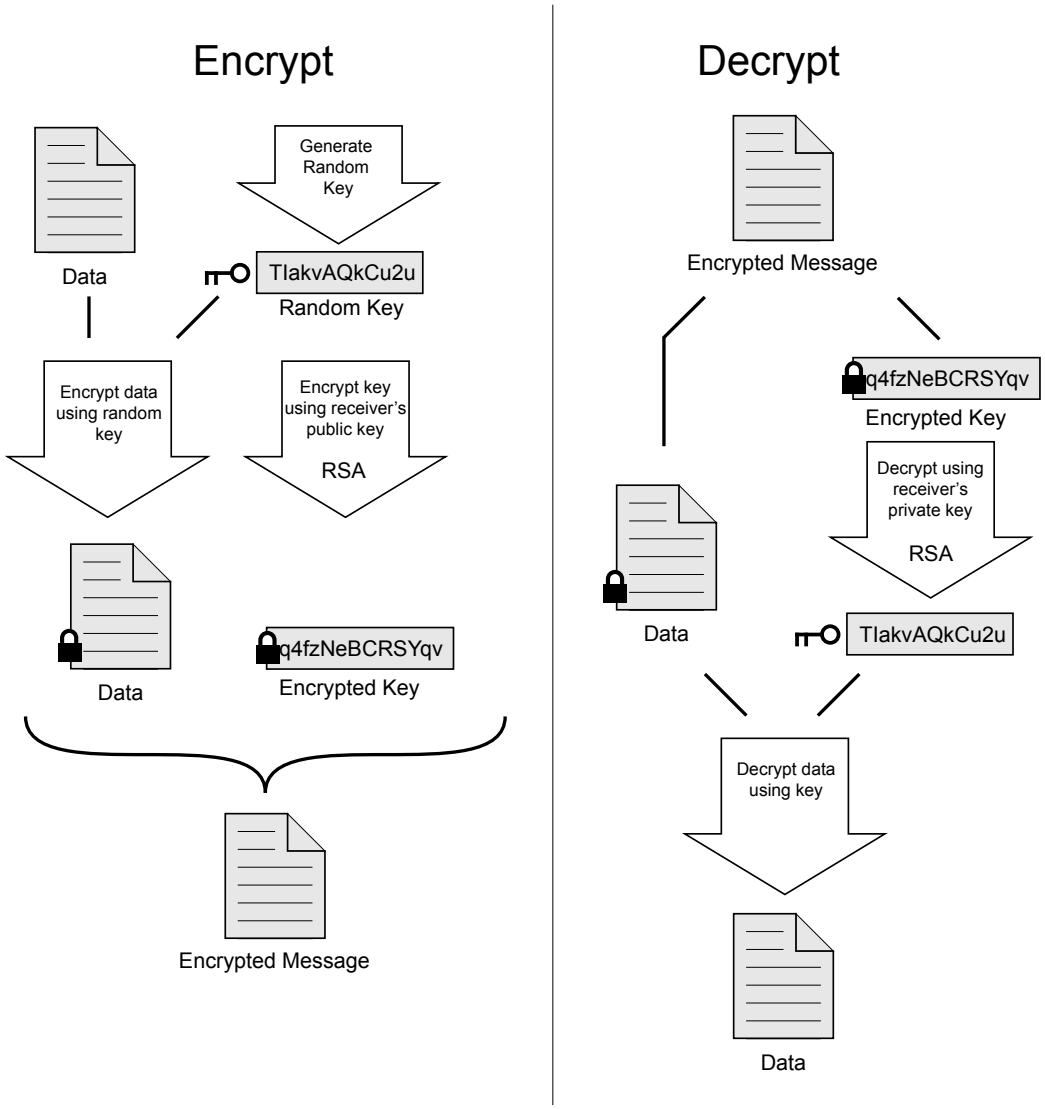

Figure 1.5: PGP encryption and decryption (no signatures) [16]

PGP uses a standard for encoding encrypted messages and keys, called ASCII Armor. This is done because some mail clients only allow ASCII characters to be displayed. So when a message is encrypted and encoded it looks like this:

—————BEGIN PGP MESSAGE—————
Version: OpenPGP.js v2.5.4 Comment: http://openpgpjs.org
wcFMA56UfPLBegUFAQ/9H+hWjRp+wJsDHEDNGCTVUxATxosvOBJuS99V2m1S
0Ux5zM/ZKuD39q8HZdTOGhclDTI4Ik55pYAtBfm7pUC+jP2Bg0HjAUC4TlIQ
Fot6wctikuTIUDxPU/6xvo9Wq+PEcaotSEd6kXdigdA0EOcNoKw+b6tQnmUV

```
YD8wvdbBsXBH+KzC8asFVGCyFd+63mUc7VcM7IOP9AalWWf/1Tq1Ad00RHzK
7MJYZt+kKa+7e/EICChF54sQQdI4YH5GjFZAxRdRSUjB4bB5mzCl/tcRcrj1
mKwS1wWj2Bvq6F5HiVLQ/jb7Ur54n2X4MbTlpgOOiBWkg1wWsU2qU/HOVVu/
A1y44uaQ+hcEyFimxcZMBveM6gdsVyou6+y97l4m0+0Gk33gITavihXdjf3e
Eumepezw4FdVokxEN7KZKImLDRyJG2S/Pule0CJJ324QZHOTrVN2RksqAC30
x0AkQRm2KMhYTRGVrn5a5T0M7WTH0U8GqzlBh/HKoQeNcDI8eRAOkeB/CR3e
yxl91Wh3F8djV4/TNVt2wxM5A5y7BiI44yeJQrw66v/zJhzax6f4wnnQpmUw
4keFd0pNPN3cCQ/g05Qkwjmy4iypM9DbLCKwyvXZhi2br/j3vaxgQYsgVRi/
BiMX1dBQXYbhXj3hNy2WA4fb6+cJyfMIojmavbshmKvUYgE71yJCTieVHjI0
Aeg8Z6tqOdAVv5R6dp7u04xyMZczdBhUSeDvQQpUdRW36wp50W8SOpBg7uX6
6Xlnpd26lWj4NMtCdnkKYfCpdSWAtRlXmROVIT/2bzntejkVHLNkMbRY=7hn
-----END PGP MESSAGE-----
```

And this is how a public key looks:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: CryptUp 4.1.6 Gmail Encryption https://cryptup.org
Comment: Seamlessly send, receive and search encrypted email

xsFNBFlBlfIBEADSyI2uJbMO8n4cUEPYnyJIQzuNIroJoS/jVzJx/pGgmsYq
M1snabzbogo3jvZWDmpCk413n7AxfMVGUVpq54RazPWuiwiPuXFxPSoA3mtD
...
D1rlvGRd2TC0GwRR3lRZImBiakRe7Y4sGKKXGsGIa7IppRyPIqpI7tHRjwtM
HePaxhzqlvb0txBYe+CLyh/Sfirf1tvwCkZOnzSGa/hgvygpK0zKBDZ6LE/1
kCvvfs2/lerrTSyz5e9AVDVodZz4TLBUoOjNL0ye7N25s5u3DrJVWQARAQAB
wsFfBBgBCAATBQJZQZX1CRDqxkc8J/SnmwIbDAAA9xgP/3JK00SmGzB5rWQa
4EKVny29UV8YkmeOQidl5TymdU23MrQ1CcWPBScU0XElgCjP0eNfjkbzYgaV
1aO+d9htUOndHMcd4cCzq6MK6TADn8CLoiVjUSWb02ebLKGvy4H4MYQ/8UHh
qFAJ43cFYkieCyw40pw6FLhz9gODZrI618K2VHUxHdvu+Mmasfmez2gi3Ui/
i+Z2iSjySRtr9uOyVFLJkX9hVPQ3bhe/ix4BcbffmOJFih+tmffo6mDRGXhd
uVTrNrJtEVrscM/JaHqmTkMEjgWzyYemd5/Nkghgjo3qSqjNziUAHUpCCrhh
P3II5oUvpHxGyfc39aRQzResXyxrM2Gsr6Ays/jOjTJ34vUipYeCgUiqABKa
LUDci//qJNygCN3UUx6w3lNaOA3NQKZa1ifkOy6j+d+5ThXbEk5hTdeW4Gdm
mX/iY2RXhF3PDlfLvYvgBbCpraVFFLRi3hOvwaQsvaUZhYhZ+oPj4ZRcWoQ3
ywi8c1kdXxEGMwOBdM5rpUG/bP94bOz9NscmyDU9sDe0gTglHTzmx17tvxpx
ASJVftr7N2Xd7kMcoUuQQ61Cx1nBIVtwICWp9YmM910bfl5KFQbZKVJCYxmt
I01+/UY1NDRacraCIeD3fhfrPGbGHqyip9fOx0aG97xteBnhJg1wFXp175P9
-----END PGP PUBLIC KEY BLOCK-----
```

The block does not contain just a public key, but can store an username, an email and other attributes. The most used type of key is RSA, and the recommended size is 2048 bit or 4096 bit, since 1024 bit keys have been cracked in 2010 (it means that an attacker found a way to derive the private key from the public one) [18].

1.3 Bitcoin basics

The Bitcoin project aim was to implement a decentralized and trustless payment system [7]. Traditional electronic payment schemes, used by banks, rely on the assumption that every accounts balances are stored on the same logic database. When money is sent from user A to user B a transaction happens: A's account is decreased and B's account is increased of the same amount. This operation must be atomic, or, in other words if just one account is changed and not the other, then the resulting transaction would be considered invalid and must be reverted.

In this scheme if user A and user B want to exchange money they have to interact with the help of a central database, acting as a middleman. They cannot communicate directly between each other, like it would happens in case of a physical cash transaction. Both users must trust the entity responsible for the central database with their money and with the information that the transaction happened. Identification, authentication and secure communication must be used to block imposters from moving other people funds or changing transaction values.

Bitcoin revolutionize how digital funds can be stored and moved. Bitcoin users do not need to trust a single entity which has absolute control over their money, but they only have to trust an open source protocol and its software implementation. Every transaction sent between users is stored in a shared database replicated by each user. The balance in each account is not stored explicitly, but rather implicitly as it can be derived from the global transaction history.

Bitcoin transactions happen in a digital currency named Bitcoin, so to avoid confusion it will be referenced in this work as BTC. BTC do not have a fixed value in term of a specific fiat currency, but since they are traded for traditional currencies (like Euro, U.S. Dollar, Yen, etc.) on exchange markets, their value is dictated by a floating price.

To send money from one account to another, a transaction message is broadcasted to every node of the network. The transaction is added to the shared database by each user only if a consensus that the transaction is valid is reached among the majority of nodes. A transaction can be invalid, for example, if a user's account does not hold enough BTC, or if two or more transactions try to send the same BTC to different accounts at the same time. The latter instance is called a double-spending attempt.

Bitcoin solves in a new way two big obstacles on the road to build a decentralized payment system: **authentication** [Section 1.3.1], **distributed consensus** and **minting** [Section 1.3.3].

1.3.1 Addresses and wallets

The traditional authentication method for financial institution relies on some trusted authority that identifies and assigns to the user some type of credential. Then the user can use the account number and the credentials (usually a password and a 2nd factor authentication device) to authorize transaction from his account.

Bitcoin does not have the concept of accounts, but rather of wallets. Wallets contains one or more public and private key pairs. The keys are randomly generated in accordance to the Elliptic Curve Digital Signature Algorithm (ECDSA) [19]. Each pair represent some sort of BTC account that the user own. In this scheme there is no central authority that check which account is assigned to which user. Compared to the traditional scheme, it is like each user randomly generates an account number each time wants a new one. There is no one to check that two users do not generate the same key pair, but the probability of doing having a collision is so small that such occurrences are deemed negligible.

A Bitcoin address, that identifies one account, is a 160-bit hash of the public part of a ECDSA keypair. An address is just a string that does not give any information about the type of the account, the identity of the owner or when it was generated, in contrast with standard banking identifiers like IBAN [ISO13616-1:2007].

1.3.2 Transactions

BTC are moved from one address to another via transactions. The technical details of a transaction would be too long for this introduction, so we will highlight just the important concepts.

A transaction is a message in a standard format that, when sent, is broadcasted to every reachable Bitcoin nodes over the Internet. The nodes are connected between each other in a peer-to-peer fashion. A transaction contains the sender address, the recipient address, the amount of BTC to be transferred and other data. Before being sent it is signed with the private key related to the public key of the address.

The ECDSA signature is the proof that the sender owns the BTC that were present in that particular address. Thanks to public key cryptography each node can then verify the signature to be sure that was indeed the holder of the address to send the funds. If the signature is invalid or the amount of BTC is greater than the amount present in the sender's address, then the transaction is considered invalid and ignored by the other nodes.

The amount of BTC present in an address at a certain point is then the

sum of every BTC sent to the address minus the the sum of every BTC coming out of the address. So to know the exact balance is necessary to know the history of every transaction related to that address. As said before, addresses are grouped in wallets, so normally users consider the balance of their wallet and not the amount present in each single address.

1.3.3 Mining and blockchain

Since each node is connected to the others in a peer-to-peer configuration and there is no central history of transactions, but that is required to know balances of each account, the most obvious problem is how to achieve consensus on the chronological order of transactions. It is in fact evident that if different transactions are processed by different nodes in different order this will lead to inconsistent balances, as each node must keep track of the whole state of the network.

For example, if Alice receives 1 BTC from Bob and then Alice sends the same BTC to Carol, then Carol has to check if the transaction is valid, and for doing so she must verify that Alice had 1 BTC available on her wallet. But if Carol has not yet received the first transaction between Bob and Alice, then she will think that Alice has no funds available, and then she will ignore the transaction.

We must consider another problem if we assume the existence of malicious nodes. The system, lacking any form of user identification and authorization, must assume that any number of malicious nodes can try to trick honest nodes to their own advantage. In a system without a solid consensus mechanism double-spending could be a common type of attack. Alice, having an agreement to buy some physical goods with both Bob and Carol in exchange for BTC, could send the same BTC to both Bob and Carol, and leaving with the goods before they notice that she has in fact double spent the same amount, so that only the first transaction accepted by the majority of the other nodes will be valid, while the other will be discarded.

Bitcoin solves the consensus problem with a proof-of-work system, also called mining (as in gold mining). Certain nodes, called miners, collect every new transaction that have not been confirmed yet. The transactions are grouped in a data structure called block. Each block, with the notable exception of the first one (called also genesis block) contains the hash of the previous block and other protocol variables. The complete sequence of blocks in the Bitcoin system is called blockchain, because every block is linked to the previous via the hash value [Figure 1.6]. The blocks to be included in the blockchain are selected with a competitive consensus mechanism. Each miner works on a hard mathematical problem, and that first that manages to

solve it can broadcast the block to the other nodes, that then check the validity of the solution and include it in their own local blockchain. The block is included only if it is considered valid, so, for example, every transaction must come from a sending address containing a suitable amount of BTC and the signature of the address owner must be valid. Once a block is included in the blockchain every node can verify its validity checking the hash, and, if valid, a consensus is reached and every transaction included in the block becomes part of the log shared between all the nodes.

The mathematical problem used as a proof-of-work is the following: the miner needs to find a valid block whose hash has a certain number of zero digits at the beginning. To change the value of the hash the miner can change an arbitrary number that is part of every block, called nonce. The hash algorithm used is SHA-256 [20], so the fastest way to find out the hash of a block is actually doing all the computation every time the nonce is changed. The miner will change the nonce until the digest of the block has, by chance, the correct number of zeroes. In the specifications of Bitcoin protocol is included that a new block should be added to the blockchain every 10 minutes or so. To achieve this objective the number of digits that must be of zero is derived from a dynamic value called *difficulty*. The difficulty is adjusted automatically every 2016 blocks (approximately two weeks). It is increased if the block generation is too fast or decreased if the block generation is too slow.

Mining is a computation intensive process required to secure the network and miners are rewarded for it. Every block contains a transaction of some amount of BTC originating from an empty address. The miner which finds valid block can redirect that transaction, also called reward, to himself. The reward was originally 50 BTC and is reduced by half every 210,000 blocks (approximately 4 years). The miners receive also transaction fees that users can add to a transaction to make it more likely to be included in a block.

In addition to the consensus problem, mining is a clever solution to the minting problem: how to distribute new money in a system without a central issuer. The miners which contribute with computational power to securing the blockchain are rewarded with newly issued BTC for their work. They also have a en economic incentive to stay honest, because they can make more money following the rules than trying to cheat with their large computational power.

Even if every node stays honest, at some point there could be two or more groups of miners working on different branches of the blockchain stemming from the same block. In this case, the group with more computational power would generate blocks faster, making their chain grow longer than the other. Every miner has an interest in converging to the longest chain, because those

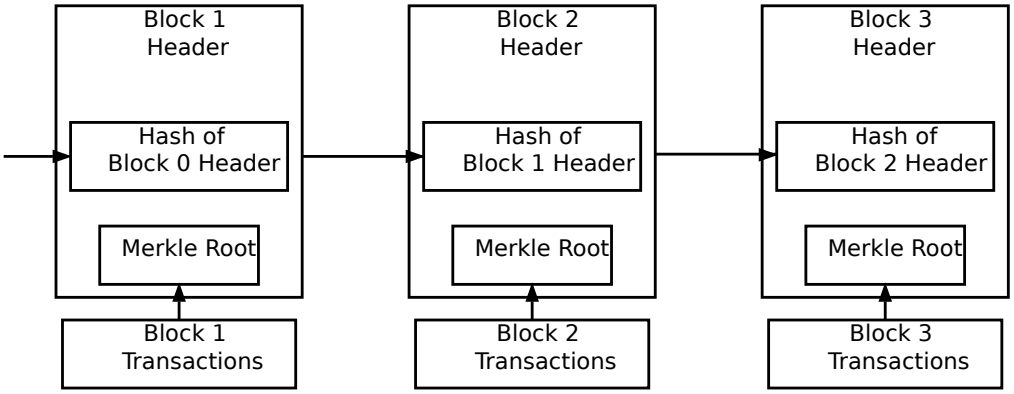

Figure 1.6: Simplified structure of Bitcoin blockchain. Transactions are stored in Merkle trees[5] for fast verification [21]

blocks will be the only ones generating a reward. In fact every other node in the system see the longest chain as the correct one, cause it received most of the computational power from miners, ignoring transactions stored on blocks part of a side chain.

Under this consensus mechanism is still possible for an attacker to hijack the blockchain for his own profit. An attacker owning more than 50% of the power available on the network (called *hashrate*) could build a chain faster than the rest of the network, crating a fork on the blockchain. In this way the attacker could modify or revert transactions already happened on the legitimate chain, opening the possibility to double spend the same BTC or to do more sophisticated attacks. Until April 2017 no such attack has been attempted, but in the future there is no guarantee that it will not happen. However, the reward per block would give an incentive for an attacker with such powerful means to stay honest and make money mining, instead of trying such an attack, that could also destroy the public trust in Bitcoin and so making drop the value of BTC.

[5]Every non-leaf node in a Merkle tree is labelled with the hash of the labels or values (in case of leaves) of its child nodes [22].

1.3.4 Bitcoin use

During the last years Bitcoin use and acceptance as a payment method has grown importantly but, in comparison with more traditional digital payment provider like credit card networks, Paypal and other Financial providers, it can still be considered a niche market. Many merchants, even big names like Microsoft [23] and Dell , accept BTC for online payments, mainly for the low cost per transaction and for the publicity that comes from using such a novel technology.

Many public exchanges allows users to buy and sell BTC with conventional currency and to practice currency trading and investment. BTC are still not widely used as a store of value, mainly because of their high price volatility and thanks to the difficulty of securing them for people without computer skills.

Libertarians and anarcho-capitalists circles have expressed appreciation for the absence of a central authority and for the ability to use a currency that is not regulated by a central bank or a government. While historically many currencies started as being issued privately by certain banks, during the 20th century almost everywhere in the world states have imposed monopolies over money supplies. Experiments with private currencies in recent years, like Liberty Dollars [24], have all failed, so some activists see in cryptocurrencies a way to finally build reliable private money.

Thanks to the use of random generated keys as accounts and addresses, Bitcoin is by default pseudo-anonymous, because it is possible to track every BTC movement following chains of transaction while not knowing the actual name of the person behind a certain wallet. In case a relationship is established between an address and a physical identity, is then trivial to identify other addresses belonging to the same individual. A way to make Bitcoin truly anonymous is using mixing services [25].

Thanks to anonymity and liquidity Bitcoin has become popular among cyber-criminals and fraudsters [26]. Various ransomwares ask for BTC in exchange for decrypting personal data and even give a discount for such method of payment [27]. Dark markets, usually hosted on Tor servers, have sprung during recent years, making easy to buy a wide range of illegal goods and services: drugs, weapons, counterfeit IDs and credit cards data, malwares, hacking and DDoS for hire services, child pornography and even assassination services [28, 29].

Bitcoin is also used to facilitate Ponzi schemes, frauds and scams. Contrary to credit cards and other payment method is not possible to revert a transaction, so when a victim of a fraud sends money it is really easy for the fraudster to just run away with the money. It makes also difficult to

identify scammers and enables them to operate from different countries and jurisdictions.

1.3.5 Limits

Bitcoin was the first and most successful cryptocurrency, but in spite of this (or maybe because of this), during the years many technical aspects have been criticized and reviewed, and improvements have been proposed, both as implementations and for theoretical problems. Some of them has been included in the Bitcoin protocol (as Bitcoin Improvement Proposals, or BIPs), while others, requiring changes deemed too radical, led the introduction of new protocols.

One big limit is the lack of versatility: the Bitcoin blockchain was built to implement a specific cryptocurrency with certain requirements (10 minutes blocks, inflation of new currency decreased by half every 4 years, pseudo-anonymity, proof-of-work as a consensus mechanism, etc.). It can be used without modifications as a notary or timestamping service, but more advanced features like Turing-complete smart contracts[6] and tokens would require developing a different protocol. There are some clever examples of protocols built over Bitcoin to achieve different objectives, as shown in section 1.3.6, but the results are usually sub-optimal.

Even without any extensions, the Bitcoin protocol enables a weak version of the smart contract concept. It includes a built-in scripting language, that can be used to send transactions that will be validated by the network only if certain conditions are met [30]. For example, one can construct a script that requires n out of m signatures before a certain transaction can be spent (such transaction is called *multisig* [31]). Such setup is useful for corporate accounts, secure savings accounts and some merchant escrow situations.

Nonetheless there are important limitations with the scripting language:

Lack of Turing-completeness While there is a large subset of computation that the Bitcoin scripting language supports, it does not nearly support everything.

Value-blindness There is no way for a script to provide fine control over amount of BTC used in transactions. As an example is impossible to limit a certain address to send no more than 10 BTC per day.

[6]Smart contracts are computer protocols that facilitate, verify, or enforce the negotiation or performance of a contract, or that make a contractual clause unnecessary.

Scalability of the number of transactions per second (tps) is also seen as an issue in the Bitcoin community. Since every transaction must be replicated in every node of the network, and most of them should be stored theoretically forever to avoid any sort of attack, a lot of space is required. On January 2017 the blockchain size has already reached 100 GB and will grow linearly with the number of transactions [32]. This could not seem a big issue for Desktop PCs, but certainly it is for mobile devices, and must be taken into account that every new user needs to download and verify the entire blockchain before being operative. The size of the blockchain could become such an inconvenience that normal user will rely more and more on remote wallet services offered by third parties, and so undermining the very idea of a decentralized network.

The reduction in the number of nodes connected to the network is a trend already happening, even if it is difficult to get an accurate estimate of the number of nodes. Network speed and computational power could also become a bottleneck, but at the moment the capacity is the most pressing issue. Many improvement have been proposed to reduce the amount of data that needs to be stored, but no one has reached consensus among developers. As a comparison, Visa (credit card network) is currently capable of processing an average of 2000 tps, while at its peak the whole Bitcoin network reached 4 tps.

Bitcoin is also critiqued for using proof-of-work as consensus mechanism. Proof-of-work, as currently implemented, requires a vast amount of computational power, and so energy, to solve mathematical problems that are essentially meaningless. Critics argue that it is a waste of energy that could be avoided using alternatives algorithms. The most proposed alternative is some sort of proof-of-stake, where the consensus is reached when users holding the majority or just part of the currency vote on the validity of a block [33]. Until now no one has found the "perfect" algorithm, but research in this area is promising.

1.3.6 Other cryptocurrencies

With the development of Bitcoin as a digital currency many started to wonder about using the blockchain technology, the foundation of it, for different things than just BTC transactions. The consensus protocol that a blockchain enables can be extended and modified for a lot of different uses. There are two different approaches for building a new consensus protocol: building an independent protocol or building a protocol on top of Bitcoin.

On-top protocol

This approach has been used to develop a number of different applications:

Colored coins The purpose of colored coins is to serve as a protocol to enable people to build their own digital currency or token on top of the Bitcoin blockchain. To issue a new currency the developer must publicly assign a "color" to a certain Bitcoin address, and then every BTC sent from that address that was already there at the time of issuance will be seen from the user of the colored protocol as a colored coin. In this way is really easy to build a new cryptocurrency without having to implement a new blockchain, a new client and waiting for adoption. Colored coins can also be linked to the property of a certain physical asset or to some shares in a company and traded as a proof of ownership.

Metacoins A metacoin is a protocol which is build on top of the the main Bitcoin blockchain, using Bitcoin transaction to store metacoin transaction. A metacoin transaction validity is dependent only on the specific metacoin protocol, so an invalid transaction will still be saved on the Bitcoin blockchain even if will be rejected by a metacoin node. This approach is not very efficient and the cost of a metacoin transaction is dependent on the cost of a Bitcoin transaction. On the other hand, it is an easy way to develop a new protocol with advanced characteristics but with low development costs since the complexities of mining and networking are already handled by the Bitcoin protocol.

Independent protocol

An independent protocol can be designed from scratch and developers are free to write their own rules specific to the domain. The first and most successful is Namecoin [34].

Namecoin is best described as a decentralized name registration database. In decentralized protocols like Tor, Bitcoin and BitMessage, there needs to be some way of identifying accounts so that other people can interact with them, but in all existing solutions the only kind of identifier available is a pseudorandom hash like 1LW79wp5ZBqaHW1jL5TCiBCrhQYtHagUWy. Ideally, one would like to be able to have an account with a name like "george". However, the problem is that if one person can create an account named "george" then someone else can use the same process to register "george" for themselves as well and impersonate them. The only solution is a first-to-file paradigm, where the first registrar succeeds and the second fails - a problem perfectly

suited for the Bitcoin consensus protocol. Namecoin is the oldest, and most successful, implementation of a name registration system using such an idea.

The independent protocol approach is more difficult to implement; each individual implementation needs to bootstrap an independent blockchain, as well as building and testing all of the necessary state transition and networking code.

1.4 Ethereum basics

Ethereum wants to solve the dilemma between building on top of an already established blockchain and designing a new protocol giving developers a framework to build decentralized app on top of a blockchain that supports out of the box Turing-complete smart contracts and the definition of new transaction formats and protocols.

Ethereum is supposed to act as foundation layer to accelerate development and reduce costs associated with building a blockchain application.

Proposed with a white paper in 2013 by researcher and programmer Vitalik Buterin, and formalized in a technical yellow paper by Bitcoin developer Gavin Wood, was lunched as a set of open source software in 2015 together with its live blockchain [35, 36]. The current version uses a proof-of-work consensus algorithm, but developers expressed a desire to switch to some kind of proof-of-stake algorithm in the future.

Ethereum has an internal currency called ether, or ETH, that con be exchanged with fiat currency, much like Bitcoin. Unlikely Bitcoin this currency does not represent the primary application of Ethereum but it is needed to act as a reward for miners and can be consumed to execute the smart contract code that runs on the blockchain.

Ethereum is a distributed system, but since it manages to come to a consensus about a single shared blockchain, we can abstract from the distributed aspects and see it as a "transactional singleton machine with shared-state" [36].

It can be thought as a state transition system, where the state is represented by the set of all Ethereum accounts and the state transition function is a direct transfer of value and information between accounts.

1.4.1 Accounts

Each account, similar to Bitcoin addresses, is identified by a 20 byte address. Usually they are encoded as hexadecimal strings and look like this `0xde0B295669a9FD93d5F28D9Ec85E40f4cb697BAe`. The address is the final

20 bytes of the hash value of the account's public key. The public key is derived from a random generated private key with the ECDSA algorithm.

An account is composed of the following fields:

- The **nonce**, a counter used to make sure each transaction can only be processed once;
- The account's current **ETH balance**;
- The account's **contract code**, if present;
- The account's **storage**.

There are two types of accounts: **normal** (or externally owned) **accounts** and **contract accounts**.

Normal accounts are controlled by whoever controls the private key, while contract accounts are controlled by their own contact code. A message containing ETH or information can be sent from a normal account to any other account when the owner broadcasts over the network a transaction signed with its private key.

On the other hand, a contract account code is activated only when it receives a message from another account. The results of the code activation can be reading or writing on the internal storage, sending other messages, or creating new contracts.

1.4.2 Contracts

Contracts in Ethereum should not be seen as something that must be "fulfilled" or "complied with", rather, they are more like "autonomous agents" that live inside of the Ethereum execution environment, always executing a specific piece of code when awakened by a message or transaction, and having direct control over their own ether balance and their own storage space to keep track of persistent variables.

1.4.3 Transactions

A transaction in Ethereum is an instruction signed by an externally owned account's private key using the ECDSA algorithm. There are two types of transactions: those which result in message calls and those which result in the creation of new contract accounts. A transaction must contains the following common fields:

- **nonce**: value equal to the number of transaction sent by the sender.

- **gasPrice**: value representing the fee the sender is ready to pay per computational step;
- **gasLimit**: value representing the maximum number of computational steps that the transaction execution is allowed to take;
- **to**: the address of the destination account, or empty in case of a contract creation;
- **value**: representing the amount of ETH to transfer from the sender to the destination;
- **signature**: the cryptographic signature of the sender

Additionally, a contract creation transaction contains also a field called **init**, which represent the new contract initialization instructions.

On the other hand a message call transaction contains a **data** field, representing the input data of the message call.

The *gasLimit* and *gasPrice* are needed to avoid denial-of-service attack and abuse of contracts by sender accounts. The sender must pay for the execution of contract code, because the code is execute by another entity and so this entity needs to get compensated for the computational power expended. The code of a contract, being Turing-complete, could also start an infinite loop and never terminate, and so the *gasLimit* ensues that the code does not keep running forever. We will specify which entity actually execute the code in the following chapters.

1.4.4 Message calls

Messages are sent between contracts and they usually trigger the execution of the code of the destination contract. Contract code, during execution, can communicate with other contracts through message calls. Essentially a message specific type of transaction, except it is produced by a contract account and not by an external account. Another difference is that a message is not stored in the blockchain but it happens just in the internal Ethereum execution environment. A message still needs a *gaslimit* because it could require the destination contract to do some work. The limit also counts recursively for every message call made by the destination contract during the same activation.

Message calls have a data field that usually contains parameters that can be accessed by the called contract. The called contract can also return data to the caller. In this way message calls are an implementation of the procedure call paradigm.

1.4.5 State transition function

As we said before, if we consider Ethereum as a state transition system, the state of the system is the set made of the states of each account (nonce, ETH value, code and storage). The state transition function accept the old state and a transaction as input end returns a new state as output.

Given a state S and a transaction T, the function can be described as an algorithm with the following steps:

- Check if T is well formed, the signature is valid and the nonce of the transaction matches the nonce of the account. If not, return an error.
- Calculate the transaction fee as *gasLimit* * *gasPrice*. Subtract the fee from the sender account balance and increment the sender nonce. If there is not enough balance, return an error.
- Initialize a variable called *gas* with the value in *gasLimit*.
- Decrement gas of a certain quantity per byte of T to pay for the storage space on the blockchain.
- Transfer the ETH value from the sender to the receiver. If the receiving account does not exist, create it.
- If the receiving account is a contract, run the contract code decrementing the gas variable for each instruction executed until it is completed or until gas is 0.
- If the gas is depleted before the contract is completed, or the amount of ETH to transfer is more than the amount present in the sender account, revert all state changes except the payment of the fee and add the fees to the miner's account.
- Otherwise, if the contract code execution is completed, refund the the fees for the remaining gas to the sender, and dd the fees paid for gas consumed to the miner's account.

The state function is described formally in the yellow paper [36].

1.4.6 Gas and fees

Gas is an essential abstraction in creating a Turing-complete general-purpose blockchain. Gas is the unit of measure of the amount of work done by an instruction or a set of instruction on the Ethereum virtual machine. Each

operation that can be performed with contract code costs a certain number of Gas, with operations that require more computational steps costing more than operations that require few steps. As an example using the instruction RIPEMD160BASE costs 600 units of Gas, while executing an ADD instruction cost 3 units. The amount of Gas per each instruction has been decided by Ethereum developers based on the theoretical number of clocks that would require on a general-purpose CPU.

We have seen that the code present in a contract is executed when a transaction or a message activates it. To make sure that someone is paying a fee for the execution of the code, so that no one can abuse the network and waste its computational power in useless operations, every transaction is responsible for paying for the execution of the code that it calls, and recursively for every call made by that code. This is the reason every message call or transaction must have the fields *gasLimit* and *gasPrice*.

The *gasLimit* is the total amount of Gas that can be consumed by the code activated during its execution, while the *gasPrice* is how much the initiator of the transaction or message is willing to pay in ETH for one unit of Gas. So the maximum fee paid is given by $gasLimit \times gasPrice$. Why creating another unit of measure instead of indicating the cost of each instruction in ETH currency? Because, with the price of ETH changing, a decoupling between the two units means that only the *gasPrice* variable needs to change, and not the nominal cost of each instruction.

Bitcoin fees, on the other hand, are linked to the disk size of the transaction, mainly because it gives an incentive to miners to include transactions in a block and not just mining empty blocks. The fees structure in Ethereum is different than Bitcoin, because the transaction size is not directly related to the amount of computational power needed to run the code, as few lines of codes with loops could be more computational intensive than program made of many lines but without any loop.

Because the contract's language is Turing-complete, it is also impossible to predict if a certain smart contract will ever terminate once activated, as we remember that the Halting Problem is undecidable for Turing equivalent machines, or how many steps it will take before terminating. So it is impossible to know the exact amount of Gas consumed before sending a transaction, and an educated guess has to be made, setting a *gasLimit* higher than that.

As we wrote in the last section, if the code terminates before the *gasLimit* is reached, the unspent ETHs are not wasted but credited back to the sender. On the opposite, if the Gas is depleted before the end of the program, the fees are credited to the miner to compensate computational resources used, but the changes made by the code are reverted, because no contract should be partially executed.

1.4.7 Code execution

Ethereum contracts are written to be executed inside a simple stack-based virtual machine (called EVM). Every node executing an Ethereum client simulates the execution of this same EVM, so that the result of the execution will be the same, no matter the architecture or the operative system in use.

The EVM executes a low-level assembly-like language (ASM). Some examples of the ASM instructions, also called opcodes, are:

```
STOP        Halts execution
ADD         Addition operation
MUL         Multiplication operation
SUB         Subtraction operation
DIV         Integer division operation
SDIV        Signed integer division operation
MOD         Modulo remainder operation
SMOD        Signed modulo remainder operation
ADDMOD      Addition and then modulo operation
MULMOD      Multiplication and then modulo operation
EXP         Exponential operation
```

During the execution the EVM has access to three different types of memory:

- A simple last-in-first-out **stack** accessible with the traditional PUSH and POP instructions.
- The primary **memory**, as an array that can be indefinitely extended.
- The contract's **storage**, as a key/value store mapping 256-bit words to 256-bit words. It's the only way for a contract to permanently save data, so that it can be retrieved in a future execution. Storage is saved publicly on the blockchain so it can be read from external applications. Encryption is the only way to achieve confidentiality for data saved on a contract's storage.

The EVM during an execution can access the global state of the network of all accounts, including ETH balances and their key/value store. It can also access the parameters of the message call that have activated the contract.

Details of EVM specifications are described in the yellow paper, but we must note that contracts are not meant to be written with this low-level assembly-like language. Various high-level languages have been developed for Ethereum programming, which are then compiled to generate ASM code.

The officially supported languages at the time of writing are:

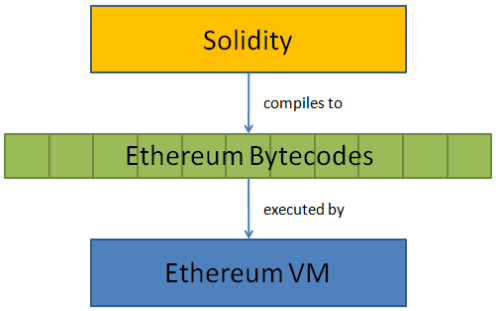

Figure 1.7: Solidity language stack [37]

- **Solidity**: a contract-oriented, high-level language whose syntax is similar to that of JavaScript and it is designed to target the Ethereum Virtual Machine (EVM).

  Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features. It supported by Remix, an Ethereum IDE, that offers a runtime environment and a compiler [38, 39].

- **Serpent**: a python-like language with domain-specific features for contract programming. The Serpent compiler is written in C++ [40].

- **LLL**: a lisp-like language, meant to be very simple and minimalistic, essentially just a tiny wrapper over coding in ASM directly. Still supported but rarely used.

Alternatively, being EVM specifications public, it is possible to build a domain specific language, with tools such as Xtext, that can be compiled to generate ASM code.

1.4.8 Blockchain and mining

As we explained in the previous sections, Ethereum relies on a global state shared by each node. The global state is saved in a blockchain, similar to Bitcoin. The main difference with Bitcoin is that in Ethereum each block contains the most recent state, other than every new transaction happened after the last block. This may seem even less efficient than Bitcoin, but actually state's data are saved in specific tree structures, called Patricia Tree, so that only new and changed data needs to be actually stored, while old data can be referenced with pointer to older blocks, without need of replication [41].

The advantage of saving the entire state in the last block is that, unlike Bitcoin, there is no need for the final user to store the entire blockchain just to keep track of account balances, saving a lot of space.

A new block is created when a miner solves a proof-of-work challenge. The difficulty of the algorithm is dynamic so that a new block is created in average every 12 seconds, instead of 10 minutes like in Bitcoin case. The really short amount of time for propagation of a valid block between nodes raise the problems of different nodes finding different valid blocks at around the same time. We will not delve into this problem, but the reader should know that this problem, common to all blockchains with fast confirmation times, is greatly reduced with a method, adopted by Ethereum, called GHOST [35, 42].

Once a transaction or the result of a computation is included in the blockchain and the block become old enough, so that a lot of other blocks are added on top of it, it is not feasible to change it without having more than 50% of the entire mining power of the network, like in Bitcoin.

Ethereum uses a different proof-of-work algorithm than Bitcoin, called Ethash [43]. The main difference with Bitcoin is that Ethash is a memory intensive algorithm, and not a computational intensive one. So the bottleneck of the algorithm is the primary memory of the node, usually the RAM on a general purpose computer. Having more memory means being able to do more work and so increasing the probability of getting the block reward.

The requirement for a memory intensive algorithm have been intentionally added to discourage the creation of ASICs (Application Specific Integrated Circuit) dedicated to Ethereum mining. ASICs have been created for Bitcoin mining, taking advantages of the fact that it is pretty easy to design a circuit that can compute SHA256 hashes faster than a CPU or a GPU. This has led to a concentration of the mining power in the hands of few big companies, posing a risk for Bitcoin decentralization [44]. So Ethash has been designed such that an hypothetical ASIC for it would require a vast amount of RAM, making the savings negligible, or the development of more efficient RAM modules, that could be reused in general purpose computers.

Another characteristic of the Ethereum mining algorithm that discourage ASIC development is that miners have to execute the code present in the block they are mining. This is done to update the state of the network, which includes the result of the computation. Making an ASIC much more efficient in executing EVM code would mean creating a more efficient kind of CPU, because, as we said before, the ASM code is Turing-complete and general purpose.

Finally, it is worth explaining where the actual computation take place. Every mining node has to execute all the code trying to find a proof-of-work

solution. Non-mining nodes, on the other hand, will execute some randomly selected message calls to validate each block every time they download a new one. This approach may look inefficient compared to, for example, traditional cloud computing, but that is the price to pay for autonomous code execution in a trustless and decentralized network.

1.4.9 Applications

Before talking about possible applications of Ethereum smart contracts, we must go back to the definition of smart contracts: a smart contract is a contract that enforces itself. Nick Szabo was the first to conceptualized them as computer programs [45]. While regular paper contracts implicitly require the presence of a judge to order one of the party to comply with the contract, and some kind of law enforcement, like the police, to physically enforce it, smart contracts is a computer program running on an hardware that can autonomously enforce those condition.

The most used example for explaining smart contracts is the vending machine. The vending machine can dispense to everyone with the right amount of money the selected product. The security of the vending machine is based on the fact that braking the safe to steal the coins would be more costly than the value of the coins themselves. The software running on the vending machine is a really simple smart contract involving a transaction between the user and the vending machine. When the right conditions are met the product is released.

So, if even a simple vending machine can implement some form of smart contract, what advantages do Ethereum smart contracts give us? The most important features are:

- **Code transparency**. In the previous example the user could not know what code was being executed inside the machine. In that scenario it would be trivial for the owner to cheat, like programming the machine to not release the product once in a while, even if the user inserted the right amount of coins.

  In Ethereum the contract's code is public, so anyone that want to interact with the smart contract can look at it and verify if there are any unwanted behaviours. So the user would not need to trust the contract developer.

- **Ability to transfer value**. Automated systems cannot directly and reliably transfer digital money (so with the exception of coins and banknotes) to and from other entities. While it is possible to use APIs to

deal with intermediaries like banks, credit card companies or payment processors, they usually require a physical person to create and operate an account, which work can not be automated (identity verification, contracts, etc.).

With Ethereum is possible to program smart contracts to transfer values, like ETH, tokens, BTC or other cryptocurrencies, directly to other entities or smart contracts, and to open and close accounts without human intervention. Another bonus is that no middleman is involved, so the parties using the smart contract know that the funds cannot be seized or stolen by any kind of authority.

Possible applications of smart contracts built on Ethereum are numerous. As a rule of thumb, every kind of problem that has been previously solved in a centralized manner for lack of a better way, but is inherently distributed, is a good candidate. Being the code execution not entrusted to a single entity or group of entities, the risk of cheating is negligible and the replication can almost eliminate any downtime. A smart contract, once embedded on the blockchain will always be executed as envisioned by its creator, without any fear of censorship or external influences.

There are many examples of applications envisioned or developed thanks to Ethereum:

- **Provably fair gambling**. Any kind of gambling based on randomness can be implemented as a smart contract on Ethereum. A smart contract can allow users to gamble ETH or any kind of token on a certain future blockchain event, for example that the hash of the next block will contain be an odd number. Users that want to bet on the event must send a certain amount to a specific address, while users that want to bet against the event will send their ETH to a different one. The sum of the two amounts will become the jackpot.

  If the event is verified the smart contract will generate transactions sending the divided jackpot to the winners. The reverse will happen in case of the event not happening, with funds automatically transferred to the users that had bet against the event.

  Being the code public, it can be inspected by everyone to verify its correctness, and being executed on the blockchain no trust is given to a single entity, like a betting provider. This gambling schemes can be labeled as provably fair.

  Similar services were already created on top of Bitcoin, like Satoshi Dice, using also provably fair algorithms. The difference is that with

Bitcoin the user needs to trust the service operator to send the jackpot in case of a winning bet. With Ethereum we have a trustless mechanism, because the contract code moves the funds autonomously.

There is however a big problem that arise if we use the blockchain as a source of randomness. Miners could affect the outcome of the bet deciding to not reveal a block they found. This way they would give up the block reward, but can be compensated in some way, either if they directly bet against the event or if they receive money from someone benefiting from their influence.

For this reason using the blockchain as a source of randomness is advised only if the possible advantage gained by an attacker is smaller than the reward of a single block (5 ETH at the moment in Ethereum) [46].

- **Escrow systems**. An escrow is a third party held fund that act as a guarantee during an economic transaction. Escrows are commonly used when two parties do not trust each-other and need to exchange large amount of money in exchange for something. They are often used in the process of buying real estates. For example, buying a house would require for the buyer to money put in an escrow held by a trusted third party, then the property transfer act must be signed by both parties, and after this the money in the escrow will be released to the seller. This procedure is done to prevent the seller from running away with the money without signing the transfer of property, and to prevent the buyer from signing the act and then refusing to pay the seller.

  With smart contracts is easy to implement an escrow system through an account bounded that release the money to the seller when each party involved in the transaction agrees, or else, after a certain amount of time, the funds are transferred to a specific account. It would be trivial to implement the aforementioned example with such system, but even a 2 out of 3 escrow could be useful. In this case only two entity would need to agree, so, even if one of the parties is malicious, the money could still be moved according to the majority.

  Ethereum has the capability to implement arbitrarily complex n out of m multiparty transactions in a very flexible way. This makes possible, between other things, to implement crowdsale and crowdfunding systems.

- **Digital markets**. Similarly to escrows, smart contracts can be used to release information instead of funds when activated by an economic

transaction. Even if the blockchain is public, it is possible to store private information on it, accessible only if certain conditions are met, through a technique called *secret sharing* [47, 48].

A smart contract could be programmed to only reveal a secret (for example a text, a key to activate a game, a password) to the user that sends to the smart contract's address the right amount. It could also use zero-knowledge proofs to show that it really holds the secret without revealing it to the user [49].

Combining these features makes possible to build a market for digital goods where the buyer does not need to trust the seller of the product, eliminating the usual controversies between parties.

- **Public records**. One of the most obvious application of blockchain technology is for storing public records. Any kind of information can be saved on the Ethereum for everyone to see. These records, thanks to blockchain properties, will be timestamped, impossible to censor and signed to guarantee authenticity.

  Example of possible applications are: identity management, web-of-trust, copyright management, real estate acts of ownership, public contracts, law archives, etc [50, 51, 52]. Lot of industries are looking into such applications, the main reason being that is often very costly to keep trusted and notarized digital data, usually involving middlemen like notaries or public institutions.

  While Bitcoin can be used to implement append only records, with Ethereum is also possible for an authorized entity to remove data from the blockchain, if required.

- **Voting systems**. Online electronic voting, with the promise of cutting costs and increasing voters turnout, has been an hot topic of discussion for a while in many countries. Letting people voting through the Internet pose many challenges in term of security, transparency, secrecy, authentication and reliability.

  Internet voting has required, until now, a central entity to verify voters' identities. If every vote must be public then voters can sign their votes and make them public on a certain platform. It is then easy for independent organizations to count and verify them.

  Secret ballot[7] is more difficult to implement, but it is usually adopted

[7]The secret ballot is a voting method in which a voter's choices in an election or a referendum is anonymous

for political elections. In order to prevent voters from voting multiple times a central entity must keep track of each voter and of each cast. In this case such an organization has the power to change the results at will. This risk can be reduced trough certification of the voting software and with public oversight, but not eliminated.

Researchers have proposed a voting protocol, Open Vote Network, that is able to combine decentralization and anonymity, built with Ethereum and zero-knowledge proof technology [53]. The solution is not scalable yet for the number of voters in a national election, but it is promising.

These are just some example of applications that can be envisioned with Ethereum smart contracts. More use cases will probably emerge as more researchers and industries start experimenting with blockchain technology.

1.4.10 Limits

Ethereum, while being a huge improvement in versatility over Bitcoin, still presents significant limits.

Proof-of-work

Ethereum relies on a proof-of-work algorithm to achieve consensus. As said before, this kind of algorithms are regarded by some as a waste of energy and resources that could be better allocated. Ethereum developers have long-term plans to switch to a proof-of-stake system. This means that nodes doing the validation of blocks do not need to show proof that some work has been done, but rather they need to prove that they have some stake in the network. An example of stake is owning an amount of ETH.

This approach would remove the concept of mining, while still make costly for bad actors to try to change the order of transactions or the state of the system. The main concern with this approach is that an algorithm that balances incentives and security, while keeping the network decentralized, is yet to be found. Ethereum developers are working on an algorithm called Casper [54], but the time frame for replacing Ethash is still unknown.

Costs and scalability

In Ethereum the user creating a contract must pay for the code execution of that contract. As an example, costs for storing 1 kB of data inside a contract can be estimated based on gas amounts per instruction in the yellow paper and gas prices retrieved with online services like Etherscan [55, 36]. Assuming

a price of 50 $ per ETH, saving 1 kB of data costs 0.64 $, with the cost per GB being 640,000 $. This huge costs are mainly the consequence of the need to replicate the data among thousands of nodes.

It is clear that using Ethereum as a storage medium for large amount of data is not viable. Storage on the blockchain should only be used to save smart contract state, while offloading other data used by the application to other platforms. Alternative methods for saving files in a decentralized way have been proposed, like IPFS [56] or Swarm [57], which still rely on Ethereum to guarantee file availability to users.

Because storage and computational costs of smart contracts are market-driven, it is an open question whether the system will be able to scale to meet user demand without becoming too expensive.

Security

There are two main security concerns regarding Ethereum. The security of Ethereum as a platform and the security of smart contracts built on top of this platform.

Regarding the former, as in every software system, design and implementation mistakes can lead to security problems and bugs. Bitcoin and other cryptocurrencies are especially vulnerable to this, thanks to the large economic incentive to compromise them (Ethereum market capitalization reached 30,000,000 $ during August 2017 [8]), and also because, being blockchains append-only, it is very difficult (by design) for users and developers to roll-back transactions in case of problems.

Ethereum, on top of this, presents a larger attack surface compared to Bitcoin. The reason is its smart counteract capability. The EVM, being a low-level abstraction, if not implemented correctly can lead to bugs that are really subtle to detect. Also, any bug found in the EVM design itself would be impossible to correct without breaking compatibility with the ASM code of smart contracts already deployed on the blockchain.

Smart contracts provide another attack vector. Being usually written in an high-level language and then compiled, even correct code could result in a defective program if a bug is present in the compiler. Unless the smart contract behaviour has been designed to be modified at run time, it is impossible to change the code already deployed on the blockchain. The DAO incident is a great example of how an obscure vulnerability has almost made possible for an hacker to steal an amount of ETH equivalent to almost a hundred million dollars [58].

[8] `https://coinmarketcap.com/`

All these security concerns mean that development of Ethereum applications should be taken very seriously and every best practice must be adopted to produce code that is secure by design. Security can not be just an afterthought.

Chapter 2

Problem definition and proposed solution

2.1 Problem: public key management and discovery

Thanks to our society being more and more reliant on digital communications relied through the Internet, the problem of securing of communications is one of the most hot topic of discussion. Without encryption over the Internet, doing many daily activity that we take for granted, like paying with credit cards or using any kind of service that require some form authentication would be too risky or impossible.

The main problem of securing the Internet, and also any activity done on top of it, is that it was originally designed without any security concern, mainly because the people who design it did not foresee the degree of popularity and pervasiveness that would have reached. No effort was spent on securing IP packets contents, because it was assumed that they would always transit through trusted nodes, like routers managed by academic or military institutions.

With the growth in popularity of Internet communication and its commercial use in the 90s, encryption and security feature were becoming essential, but instead of building a new protocol from the ground up, that would have required an enormous investment and the loss of backward compatibility, people in charge of it decided to try build secure applications on top of an insecure platform. This decision, probably good for the overall development of a global network and for the adoption of many shared standard (TCP/IP, SMTP, HTTP, etc.), had on the other hand made life more difficult for developers to implement ways of exchanging information without

fear of eavesdropping or interception by a third party.

An example of this is email, that, in the original form, used the SMTP (Simple Mail Transfer Protocol) to send plaintext data between users. Encryption and digital signing has been added to the standard, but only covering a part of the journey, like between the client and the SMTP server, or between two SMTP server, without any end-to-end solution. Making SMTP secure is difficult because many parties need to coordinate and adopt the same security standard, even the ones that have nothing to gain, like service providers.

To solve this problem, one of the most popular tool used is PGP. Pretty Good Privacy is an encryption program created by Phil Zimmermann in 1991 to encrypt, decrypt and sign texts, emails, files, directories and disk partitions [17]. As explained in section 1.2.3 PGP uses an hybrid between symmetric and asymmetric encryption. Thanks to this is possible to achieve end-to-end encryption over plaintext email following the few steps:

- Alice obtains Bob's public key.
- Alice uses PGP to encrypt the plaintext with Bob's public key.
- Alice sends the encrypted text over conventional email to Bob.
- Bob decrypts the encrypted text with his private key using PGP.

This way the security of the email provider or the protocol does not matter, because only Bob can decrypt the email. Also Alice can sign the email with her private key, this way, if Bob knows her public key, he can verify that the message was actually sent by Alice.

It is clear that following this scheme the first step is also the most critical: how Alice can obtain Bob's public key. Public key management is a problem as old as asymmetric cryptography, and many solution have been proposed. In this case, for PGP keys, two different solution have been proposed, **web of trust** and **public key infrastructure**.

2.2 Web of trust

Web of trust is a decentralized system to establish the authenticity of the link between a public key with its owner identity. The concept is based on the idea that each user will sign the public keys of the other users that he trust, and then publish them. In this way keys are linked together and if the identity of a user is trusted than is reasonable also to trust the identity of the other users that he vouches for.

The web of trust concept was first introduced in the cryptography environment by PGP creator Phil Zimmermann in 1992 in *The official PGP user's guide*:

> As time goes on, you will accumulate keys from other people that you may want to designate as trusted introducers. Everyone else will each choose their own trusted introducers. And everyone will gradually accumulate and distribute with their key a collection of certifying signatures from other people, with the expectation that anyone receiving it will trust at least one or two of the signatures. This will cause the emergence of a decentralized fault-tolerant web of confidence for all public keys.

With this scheme Alice can ask Bob's public key to Carol, Chuck and Dave, being users that Alice trust and that she has verified their respective public keys. They sign Bob's key with their public keys, and Alice can decide how many confirmation she needs to trust Bob. It is worth noting that this process is automatic, because every time two users communicate using PGP with web of trust enabled, a list of every trusted contact is attached to the email and updated by the recipient.

Usually key ownership between users that trust and know each other is verified in person, sometimes at so called key signing parties. Under this scheme the decision about who to trust is left to each person and so the control is more granular. The decentralization leaves out governments and big corporation that can not influence the system, and if someone's key is compromised it can usually be revoked, at least if the original user still have the private key.

Thanks to the Small World Theory [59] the average path lenght between users in a web of trust should be pretty small, giving the ability to communicate securely even between strangers that never verified their respective keys.

The biggest problem with web of trust is that, among PGP users, almost nobody uses it. This conclusion is based on an analysis of the strongest set: [60]. There are different reasons why the adoption of a web of trust concept is not bigger.

It is not practical to meet people in person, and signing parties are extremely rare outside the cryptography circle, for obvious reasons. Moreover there is no practical incentives for users to add new trusted key to their key-ring, but must be done only as an altruistic effort.

Finally there is the possibility for users to lose their private key, such that they have then no access encrypted messages sent to them. In this case it is

also impossible to create a revocation certificate, and the public key remains on the web of trust until every user that trusted it decides to revoke his trust. For this reason the decision of contacting a person using a public key can not be based only on the web of trust, but it must be confirmed in some other way, or else it is possible to send a message that the recipient will never be able to read.

2.3 Key server

The other method that user have to obtain public keys is through key servers. These servers are usually handled by organizations, such as PGP Corp [1] and MIT [2], or by individuals as a pro bono service, and they are reachable through the HKP (HTTP Keyserver Protocol) or a web interface, as shown in picture 2.1. They associate an email address with a public key or with a key ring (collection of public keys), and makes them searchable. Everyone can add an email and a public key, but only the owner can revoke his own key. Some servers also verify the ownership of the address sending a confirmation mail.

Key servers can be used as a complement of web of trust, if users find public keys that are not confirmed by their network of contacts, or they can be the only trusted source for keys. In the latter case there are some security drawbacks.

The entity which controls a key server has complete control over it, and it can change public keys and addresses associated to a certain user at will. Even if the organization that controls it can be considered trusted, like a University or a NGO, nobody can guarantee that it will not be compromised in the future. Even more worrying, an attacker which gains control of a key server could choose to show a different public key only to certain user, so that the owner would never discover the anomaly just judging a server response.

In this case when a user want to communicate securely, other than trusting the current PGP software implementation, his machine and the recipient's machine, he also need to trust that the key server is giving him the right public key, adding a weak spot to the concept of end-to-end encryption.

Finding a solution that eliminates or reduces the amount of trust that users of PGP solutions need to extend to the key server would greatly reduce the attack surface, increasing the security and reliability of secure communication through email and other channels

[1] `https://keyserver.pgp.com/vkd/GetWelcomeScreen.event`
[2] `https://pgp.mit.edu`

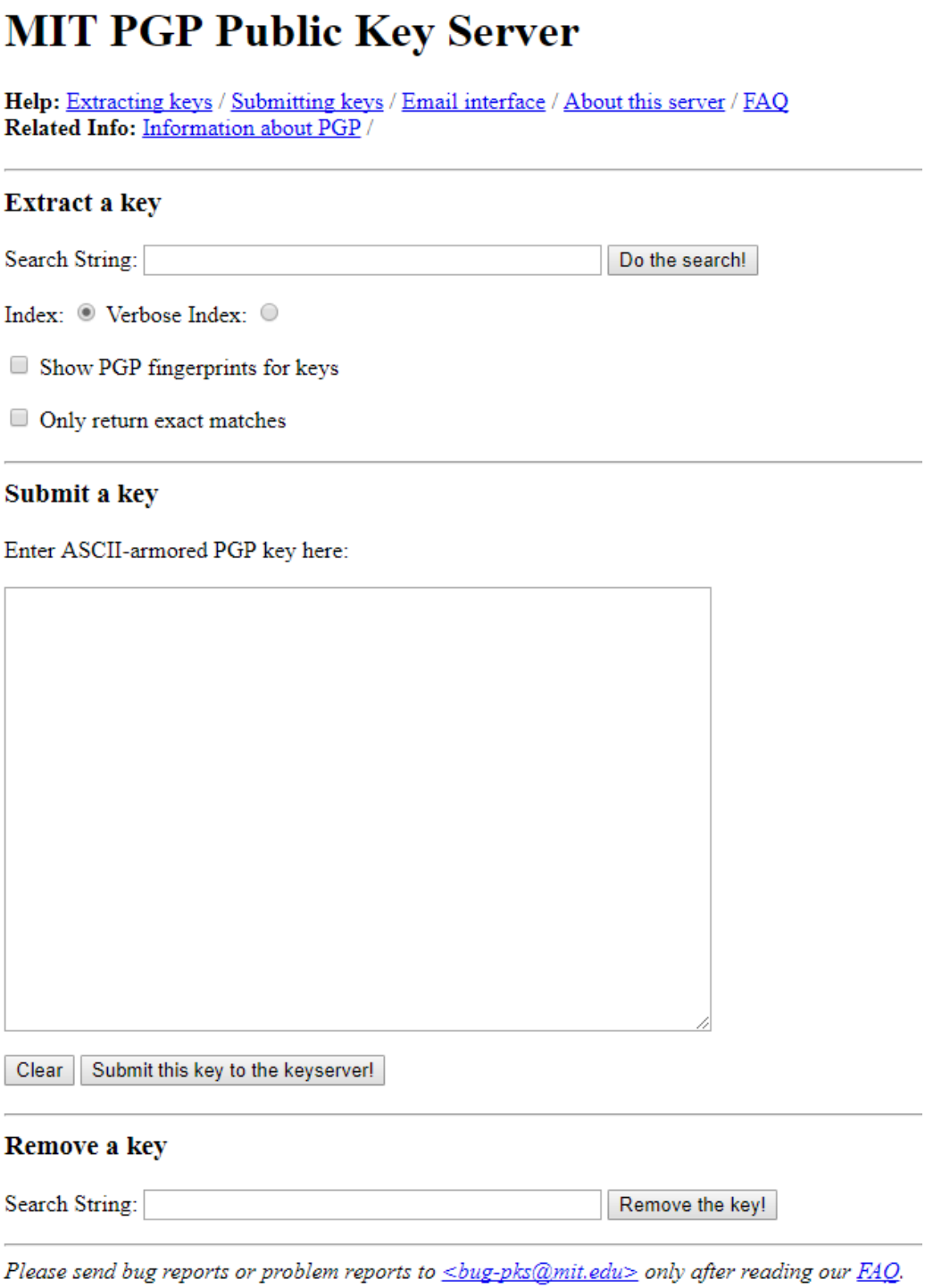

Figure 2.1: Web interface of the MIT Public PGP Key Server.

2.4 Proposal: Ethereum as key server

The aim of our proposal is to realize a web application able to verify and store on the Ethereum blockchain data like PGP public keys, email addresses and other contact information, like social accounts or domain addresses, instead of using a traditional key server. In this way, thanks to the immutable properties of the blockchain, once the data is written on it, we can expect that no attacker will be able to change or delete it. The only way would be revert the history of transaction and creating a new branch of the chain, but this would require an incredible amount of resources and expertise.

To use a metaphor, storing public keys used for communication on the Ethereum blockchain is similar as keeping the car key's inside of a military submarine: no one that has the means to steal a submarine will be interested in your car's key, and even if he would, the probabilities that he would just take the keys and returning the submarine without anybody noticing are incredibly small.

The are still risks involved with this approach: one is the longevity. A blockchain is still useful only if adopted by consistent number of users and miners. Bitcoin has been running since 2009 and it is still gaining popularity, showing that blockchains can have a life expectancy of a decade or more.

There are usually two ways in which a blockchain can "die": a catastrophic failure, like a bug or error in the protocol, that could prevents new block generation. If the bug is not corrected fast enough users could lose confidence in the network and stopping using it en masse. The other way is through a slow death, loosing users and miners little by little.

In both the cases, people relying on Ethereum could simply switch back to other methods of key discovery without risks.

Storing data is not the only duty of our proposed application. The other, even more important function, is verifying those information. This means checking that the user owns the private key corresponding to the public key, and that he controls the email address, the domain an the social account. This can be achieved in various ways but first an important decision must be taken: having the verification logic **on-chain**, in a smart contract, or **off-chain**, in a normal server.

In the following sections we will assess the feasibility of each solution.

2.4.1 On-chain verification

Having the verification process running inside a smart contract on the Ethereum blockchain would be, apparently, the best choice. In this way users only need

to check the public code of the smart contract and trust the Ethereum network, without trusting any third party.

Unfortunately using a smart contract for verification is problematic for two main reasons. The first involves signature validity verification. Users could verify the ownership of the public key sending to the contract a transaction containing a message signed with the user public key. The message would include the contact information to be saved on the blockchain. The smart contract could simply execute an algortihm to verify the signature. The problem is that currently verifying the kind of cryptographic signatures used by PGP (RSA, DSA and Elgamal) is too expensive to be done on-chain and, even if it was cheap, there are no libraries available, requiring to implement it by scratch using solidity or some other language. Of course it is never a good idea to implement your own cryptographic algorithm if you are not an experienced cryptographer.

There are other possible solutions to this problem, but they are not currently viable:

- Using an *oracle*[3] (aka. external library) to do signature verification, but then relying on a trusted third party.
- Waiting until the next release of Etheremum (Metropolis), that will allow RSA cryptography to be implemented efficiently [61].
- Waiting for EVM2.0, the next generation of the Ethereum machine, which would provide near-native performance for contracts. RSA could be written in a contract efficiently [62].

The second problem with doing verification on-chain is that the a smart contract can not directly operate outside the EVM environment. This means that there is no way to send emails, connect to domains or use social networks APIs. These are all operations required to verify contact information. For example, to verify that the user control a certain address the smart contract could send random code through email, and then ask the user to enter it. Without delving into the hurdles of generating a random code on a blockchain and keeping it secret, while both code and execution are public, there is no way for Ethereum to send an email in a trustless way.

The only user information that can be checked directly on-chain is the Ethereum address. The user could be asked to send a transaction from his address to the smart contract to verify the he owns it.

Mails can only be sent from an SMTP server and there is no way to for a blockchain to interact with an external server, without having to trust it.

[3] `https://blog.oraclize.it/understanding-oracles-99055c9c9f7b`

Making a network of oracles is a concept born specifically to solve this kind of problem, but unfortunately not yet available. The same also applies for every other kind of interaction outside the blockchain environment.

To sum up, verifying contact data using just a smart contract is not yet feasible yet, and it will not be in the future. The only exception would be using a network of oracles, if this concept will evolve in a way that requires less trust than doing verification off-chain.

2.4.2 Off-chain verification

With off-chain verification, contact information is first verified from a regular server and then sent to the blockchain for storage. The verification is done using different methods to check that the user has control over email address, key, domain and social network account. Then the information will be embedded in a transaction sent to the smart contract from an Ethereum client running on the same server.

Because the transaction is sent from our wallet, we will have to pay the cost to update the smart contract with contact data. To make the user cover this cost we could require a payment in ETH before starting the verification phase. Even better, remembering that we need also to verify the Ethereum address of the user, we could solve two problems at once: the user will send a transaction to pay for the cost directly to the smart contract, with a verification code attached showing that sender is indeed the owner of the other contact information.

We will delve into the specific nature of the verification process in the next chapter. The important thing now is understanding the implication of choosing to have the verification process handled by a regular server rather than by a smart contract. Our server could indeed be breached, or we could betray the trust of our user, changing their contact information at our advantage. For example we could put on the blockchain a public key controlled by us, instead of the real one, and use its private key to decrypt intercepted emails. Another thing an attacker with access to the verification server could do is registering bogus data for an unaware user, that will probably not notice it for a long time.

We must acknowledge that these risks are the same as using traditional key servers. The advantage of storing data on the blockchain with off-chain verification is that once the information is added to the smart contract, it is impossible to modify or delete it. Transparency is also increased, because the user can verify with different Ethereum clients or third party services that information is correct, while with a centralized server the user is never sure of which keys and accounts are showed to other users. Thanks to the extensive

data replication, fundamental concept of Ethereum, it is very unlikely to have any kind of downtime or information loss.

In the following scheme we try to summarize the pros and cons of each solution.

	Key server	**Smart contract (off-chain verification)**	**Smart contract (on-chain verification)**
Server required	Yes	Yes	No
Editable by attcker	always	before publishing	never
Data can be taken down	Yes	No	No
Replication	No	Massive	Massive
Costs	Admin	User and admin	User[4]

In conclusion, we think that we can implement a new concept of autonomous key server based on Ethereum smart contracts, that can improve security and convenience of key discovery, to achieve secure communication using PGP encryption. Moreover, after a careful analysis, we conclude that, to reach the best compromise between security, convenience and costs, the best solution is verify user submitted information off-chain, on a properly designed server.

Chapter 3

Design

3.1 Requirements

The aim of this thesis is to realize a system, composed of a web application, a server application and an Ethereum smart contract, able to verify, store, revoke and display user submitted contact information. Users must be able to access the web application through a normal web browser and submit their information through a form. The system must be compatible with as many different web browser and operating systems (Linux, macOs, Windows, etc.) as possible.

The user contacts information will be stored as data units, called also **contact cards**, containing the following fields:

- **Ethereum address** Text string representing an Ethereum address as defined in section 1.4.1. The address will be used also as the unique identifier of a contact card, so that the same address cannot be used two times.
- **Email** Text string representing an email address.
- **PGP public key** Text string representing a PGP public key. The key must be ASCII Armored according to the OpenPGP standard, as explained in section 1.2.3.
- **Facebook Id** Text string representing a unique identifier of a Facebook profile.
- **Domain address** Text string representing the URL of a domain.

Each field is mandatory and the user must be able to verify the ownership of each one.

The system must support three main use cases: **submission**, **look-up** and **revocation**.

Submission

The submission use case is divided into three phases: verification, payment and storage. During the first phase the user must enter the contact information on the web form and verify the ownership of the Ethereum address, email address, domain address, Facebook account. He must also provide a PGP public key and prove the ownership of the related private key. Every of these fields are mandatory and is not possible to submit just a subset of them. The application should check, before attempting verification, if the given Ethereum address has been already associated with a contact card and return an error if that is the case, since the Ethereum address must be the unique identifier for each data unit.

During the payment phase, once everything is verified, the user must pay a fee to compensate the application provider for the cost of uploading data on the smart contract. The payment must be fulfilled in ETH through a transaction from the user's wallet. Once the payment has been received the application must upload the contact card on the smart contract for storage.

Look-up

The user must be able to look-up an Ethereum address to find the relevant contact card. The application must provide an input form to insert the address. If the address is present on the smart contract the website must show the relevant information. If a contact card has been revoked it should be highlighted. The same information should also be publicly accessible querying the smart contract, using blockchain explorers like Etherscan[1] or other tools.

Revocation

The user must be able to mark a contact card as revoked. Only the uploader of a certain contact card must be able to revoke it, so some sort of verification must be implemented. Once a contact card is revoked it should be highlighted at every look-up.

[1] `https://etherscan.io`

3.2 Logic model

After analyzing the requirements we have built a logic model for the proposed system. It describes the system in a formal way without specifying the actual implementation. The model is represented over three dimensions: structure, interaction and behaviour. The structure model defines the different parts of the system and the external entities it must interact with. This model is purely architectural, and does not describes the interaction or behaviour of the single entities.

The interaction model describes chronologically the interactions happening between different entities during each use case. The most important aspect of the model is the order of these interactions and their types (synchronous or asynchronous, message, request response, etc.).

The behaviour model describes how the whole system behaves during each use case, from the perspective of the user interacting with it. It does not describe the inner working of each component, which will be discussed in the implementation chapter.

3.2.1 Structure

The system is distributed between different machines and it is composed of three subsystems: a **web application**, a **server application**, and a **smart contract**. Each of these parts is modeled as different entity.

We decide to model another component that is not directly part of our system: the Ethereum wallet controlled by the user, because it is integral to the submission use case. We also introduce two actors: the user, that is able to register and revoke his own contact card, and the spectator, that can look-up the information thorough the web application or directly on the blockchain.

The entities of the system based on which node they are running on: user machine, server machine and blockchain. The diagram in picture 3.1 describes the structure of the whole system. The entities of the system are grouped based on which node they are running on: the user machine, the server machine an the Ethereum blockchain.

User machine

The web application is an active entity running inside a browser, on the user machine, and has both an internal logic (that will be probably implemented in JavaScript) and a user interface (that will be probably implemented in HTML and CSS). As most browser application, the code does not reside

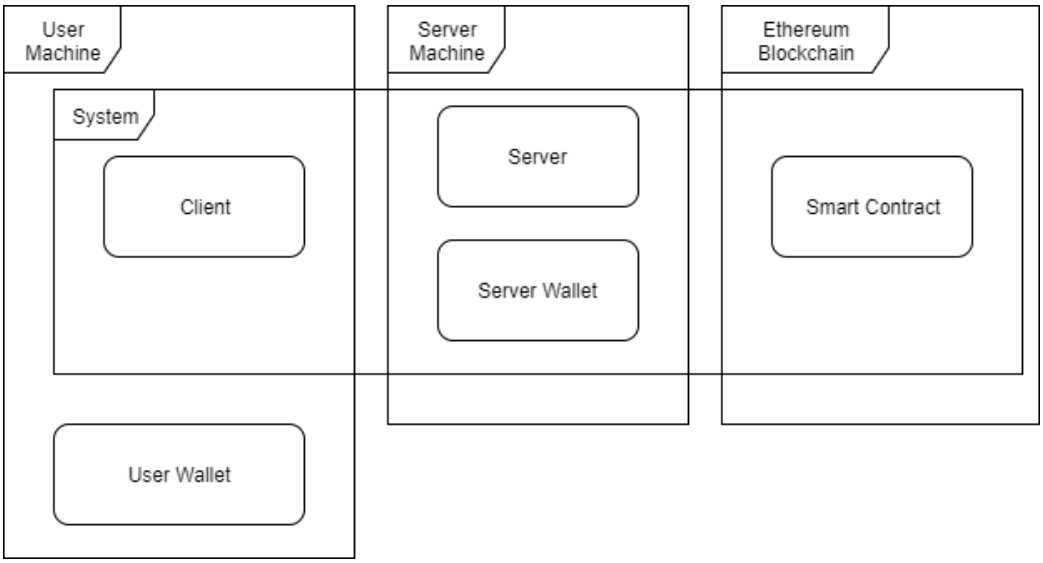

Figure 3.1: Structure model

permanently on the machine, but it is downloaded from the server every time the user needs to use it.

The web application must be compatible with a standard web browser, like Google Chrome, Mozilla Firefox, etc., but also with an Ethereum Dapp browser, like Mist or Metamask. It is divided in different standalone components, called pages. Each page handles different steps of the use cases. The distribution of different functionalities in each page will be discussed in the implementation page.

Another component, not part of our system but included in our model, runs on the user machine: the user Ethereum wallet. The wallet can be hosted directly in an Ethereum Dapp browser, like the ones mentioned above, or in a simple Ethereum client, like Geth or Parity. The difference is that with the former the web application can generate an Ethereum transaction directly inside the web application and then ask for user authorization through the browser interface, while with the latter the user must leave the application, create and execute the transaction on a different program, and then going back to the web browser.

The wallet is essential in our model for two reasons: to verify the user's Ethereum address and to pay for storage costs on the blockchain. It is then required for the user to have his own client installed on the machine before using our application. If the user does not have one, he can follow the instructions that will be provided for downloading and installing a client

from a reputable source.

Server machine

The server is an active entity running the server machine controlled by the application provider (us). This entity is composed of two parts: the server application, running inside a web server, and an Ethereum client. The server application handles the interaction with the web application through HTTP request and responses.

The server application is the core of our system, and has the responsibility to verify user submitted contact information and to handle most of the business logic. It cannot however interact with the blockchain, and so we need an Ethereum client for this purpose. The server application can interact with the Ethereum client through a local API, since both run on the same machine.

Blockchain

As we explained before, smart contracts are executed concurrently by miners, and their results are propagated when a miner finds a valid block. For the scope of this model, we can abstract away the details, and consider the Ethereum blockchain as a single separate machine, with a smart contract running on top of it.

The smart contract is modeled a single passive object that can only react to incoming transaction that activate specific procedures. Procedures can read or modify its state, which is publicly visible on the blockchain. Another important characteristic of the smart contract is that it can receive, hold and send ETH, acting like a wallet.

3.2.2 Behaviour

After analyzing the structure, we must describe the behaviour of the system, as seen from the user perspective. We have done that through activity diagrams that explain each use case.

Submission

During the submission use case the user interact with the system with the goal of uploading his public key and contact information to the blockchain. A general model of the behaviour of the system is presented in picture 3.2.

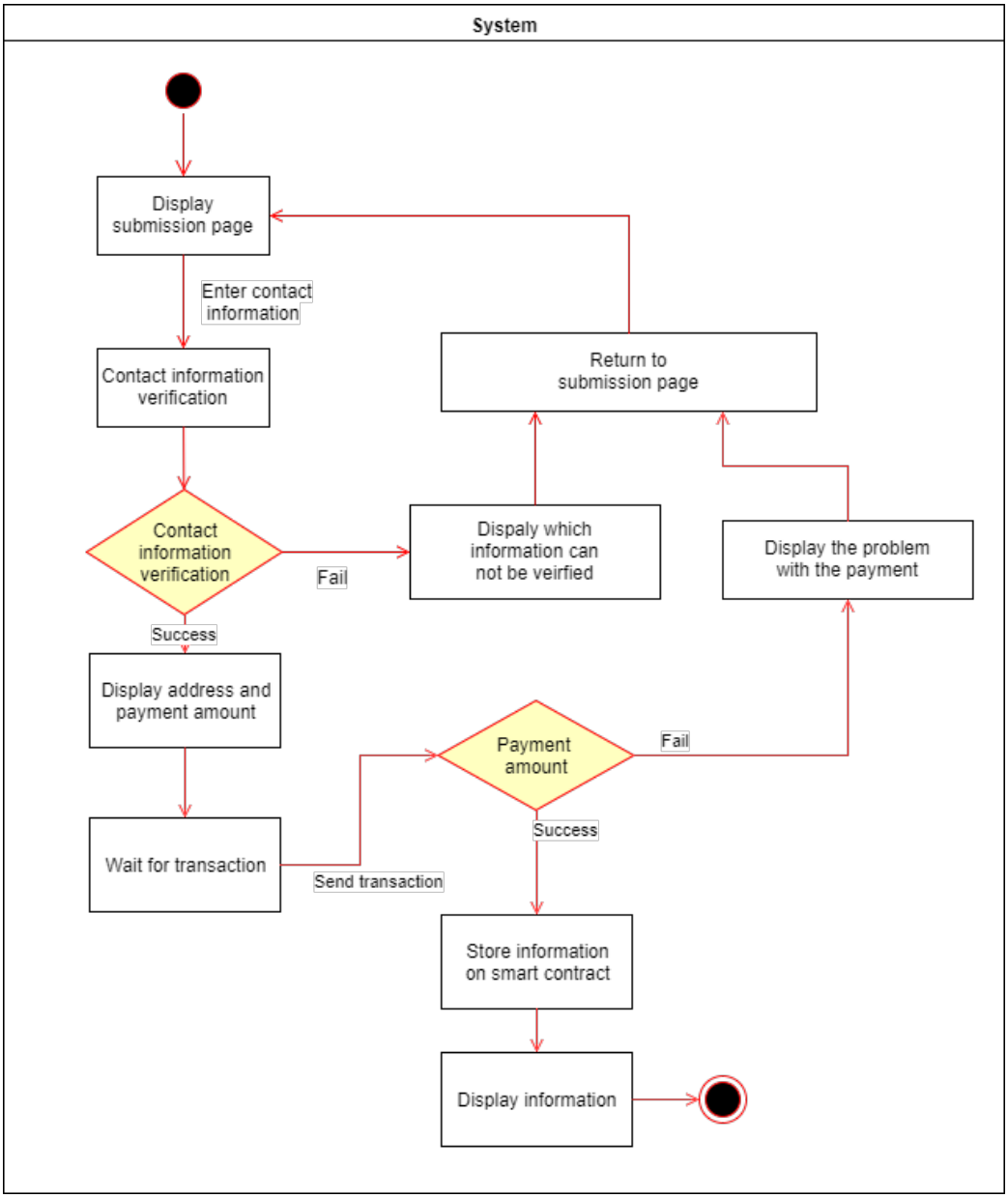

Figure 3.2: Submission activity diagram (general)

In the model we can see how the system as a whole behaves during the submission phase. This diagram gives us a general understanding of the use case, but it is not detailed enough, especially during the verification phase.

As we explained in the previous chapter, a way to verify the Ethereum address is for the user to send a certain amount of ETH to the smart contract. In this way the user could be verifying that the user control the address and paying for the storage fee at the same time. So we decided to group the verification phase and the payment phase in the same activity.

We have also defined the verification process of each contact information.

- **Email address** and **PGP public key**. A random code is generated by the server, it is PGP encrypted using the user's public key and then it is sent through email to the given address. The user must then decrypt and enter the code on the web interface, proving to the server the ownership of the address and of the private key related to the submitted public key.

- **Facebook Id** The Facebook Id can be verified through the official API. At first the client-side interface shows to the user the official Facebook log-in interface. The user must then enter his credentials to authenticate with Facebook. Once the user is authenticated, the client-side application obtains from Facebook a string called *AccessToken*, that grants the access to the user id through the API. Then, the *AccessToken* is sent to the server application, which can retrieve the Facebook Id directly from the API. In this way the server can obtain the Id directly from Facebook, proving its authenticity.

- **Domain address** A random code is generated by the server, embedded in a Json file, and downloaded by the user. The user must then upload the file on his private server. The server application will then verify through an HTTP request that the code is indeed reachable at the given domain.

- **Ethereum address** A random verification message is generated, partially based on the other contact information. The message, the smart contract address and an ETH amount is then presented to the user. The user must send a transaction to the given address, with the given message attached and the correct amount of ETH to pay for the storage fees.

A more detailed model of the submission phase is presented at figure 3.3, which reflects the fact that the verification phase and the payment phase can not be kept separate. It highlights also each different verification process.

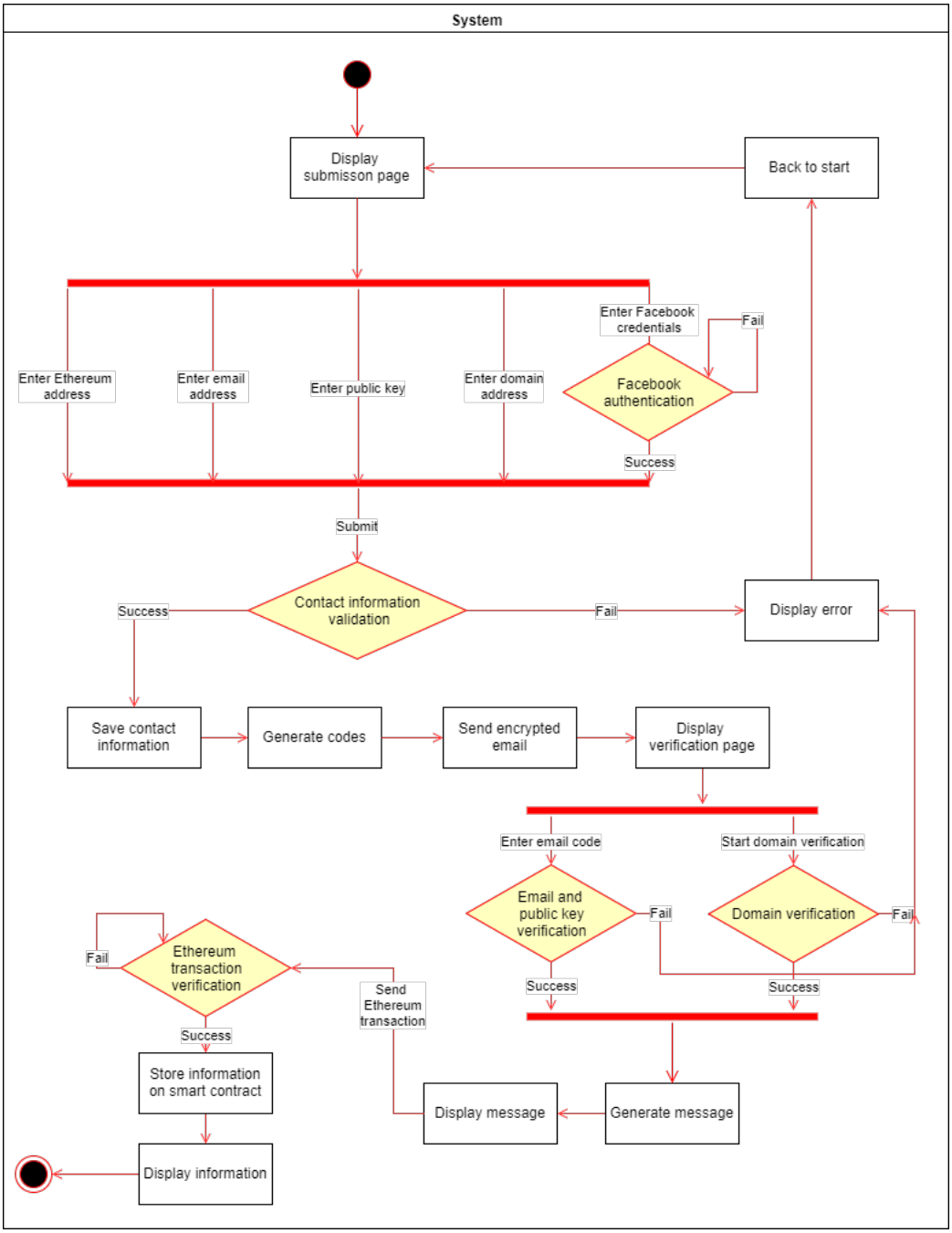

Figure 3.3: Submission activity diagram (detailed)

Look-up

During the look-up use case the user interacts with the website to check if a certain Ethereum address is associated with a contact card. If that is the case, the contact card is showed in the web interface. The contact card is displayed with a warning if the revocation flag is set. The activity diagram representing the look-up use case is represented at figure 3.4.

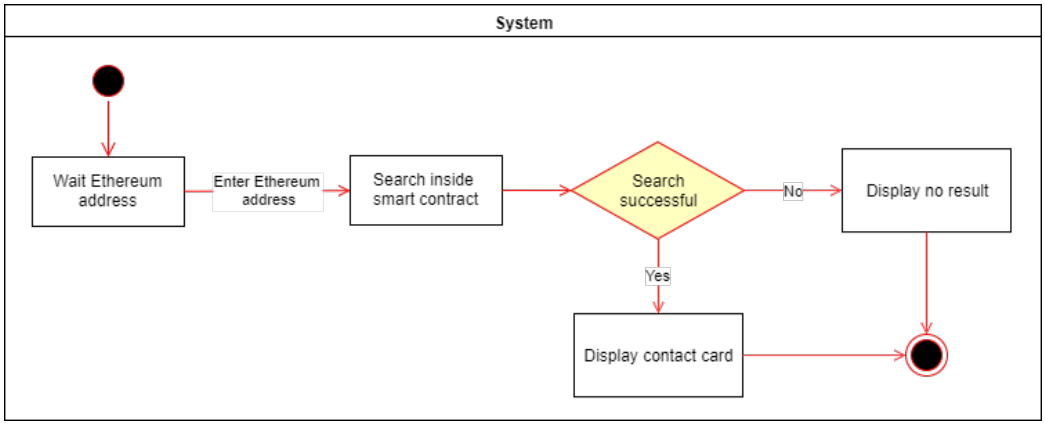

Figure 3.4: Look-up activity diagram

The user can also access the contact cards independently from our system, querying directly the smart contract. This can be achieved with a blockchain explorer service, like `Ethercast.io`, or by installing an Ethereum client on the user machine.

Revocation

Through the revocation functionality, the user can decide to revoke a contact card already stored on the smart contract. To revoke a contact card the user must be verified as the owner of that information. During the design of this use case we had to decide what constitutes a proof of ownership of the contact information. There were two options: the Ethereum address or one of the other contact information (email, public key, domain, Facebbok id).

If we use one of the other contact information, the server should verify again the user and then send a transaction to the smart contract, calling a method to revoke the contact card. This has two main consequences: firstly, if an attacker takes control of the contact information required for the revocation, he can easily revoke the contact card and secondly, the server has to be trusted again, like in the verification process. This means that if the

service provider decides, for any reason, to stop supporting the service, the user would not be able to revoke the contact card.

The other strategy involves letting the user that controls the Ethereum address of the contact card do the revocation. In this scenario the user would send a transaction from his wallet to the contract address, calling a specific procedure that then sets the revoked flag of the contact card. The smart contract would check if the transaction has been originated from the same address included on the contact card, verifying the identity of the user. In this way the user does not interact with, and does not need to trust, the server application.

We decided to adopt the second process, mainly because we think that the user has more or less the same probability of losing his PGP private key or his Ethereum private key, but in this way the user can revoke his contact card even if the server is not reachable or the service has been discontinued.

As a way to make the application more user friendly we introduce the possibility for the user to generate a revocation transaction directly using the web application. The only requirement for the user is to visit the page with a Dapp browser like Mist or Metamask. The web page would prompt a transaction using the user's wallet, so that the user only need to authorize the transaction. Both scenarios are represented at figure 3.5.

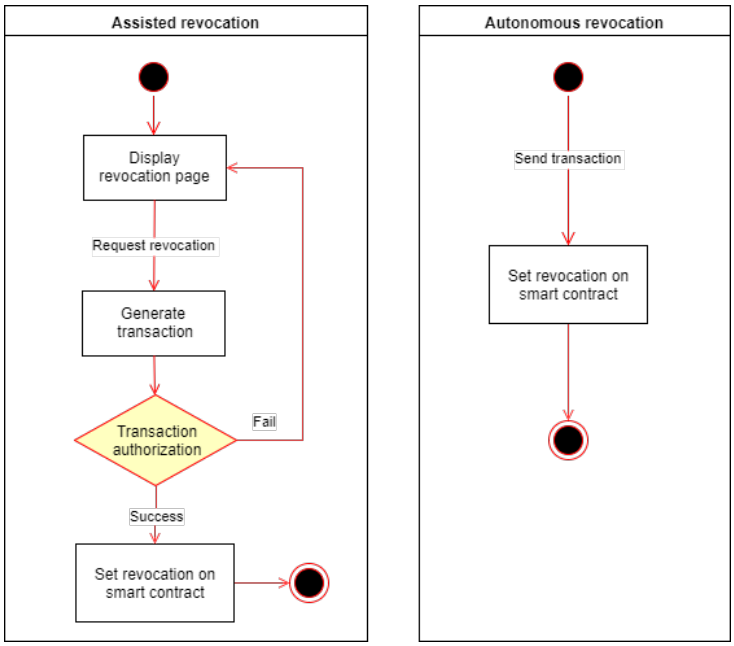

Figure 3.5: Revocation activity diagram

3.2.3 Interaction

Being the system composed of three subsystems (web application, server application, smart contract) interacting with each other, having a good interaction model is essential to get a real understanding of the work required to implement the final application. In each of the following sections an interaction diagram will be presented to model each use case. The interaction between different machines always happens over the Internet, but using different communication patterns. Email and Ethereum transactions will be considered asynchronous messages, while HTTP request and response as synchronous calls.

Submission

Submission is the most complex use case under the interaction perspective. For this reason, firstly we have designed a general diagram that gives simplified understanding of the process, presented at figure 3.6. In this diagram the system is composed of the three main entities, running on three different machines, interacting with each other.

Two interaction patterns are represented at this abstraction level: synchronous request-response and asynchronous Ethereum transaction. HTTP requests are modeled as synchronous because we assume that the entity doing the request waits for the response before doing anything. The nature of an Ethereum transaction is asynchronous, because this is how they behave on the Ethereum network. If we want to check the state and the result of an ethereum transaction we each time need to check the state of the blockchain. The only exception is represented by the *Message and amount* request and the *Message and amount* response: this interaction is in fact happening between the server and the Ethereum client, and so, being a read-only operation, does not require sending a transaction on the blockchain.

The second diagram, displayed at figure 3.7, describes in greater detail the interaction between the user and each part of the system, not just the three main entities. While in the previous diagram we considered just the machines as single entities, in this model we split them in sub-entities, to be able have a more accurate model at a lower level of abstraction.

We also introduced the user, as an actor, and two types of interaction, both considered asynchronous messages: *User input* and *Email*. A user input interaction happens when the user gives some kind of input (information or command) to a certain entity. An Email interaction happens when an email is sent form the server to the user's email address, to verify it. We decided to not model the email server and the PGP software, but, for the sake simplicity,

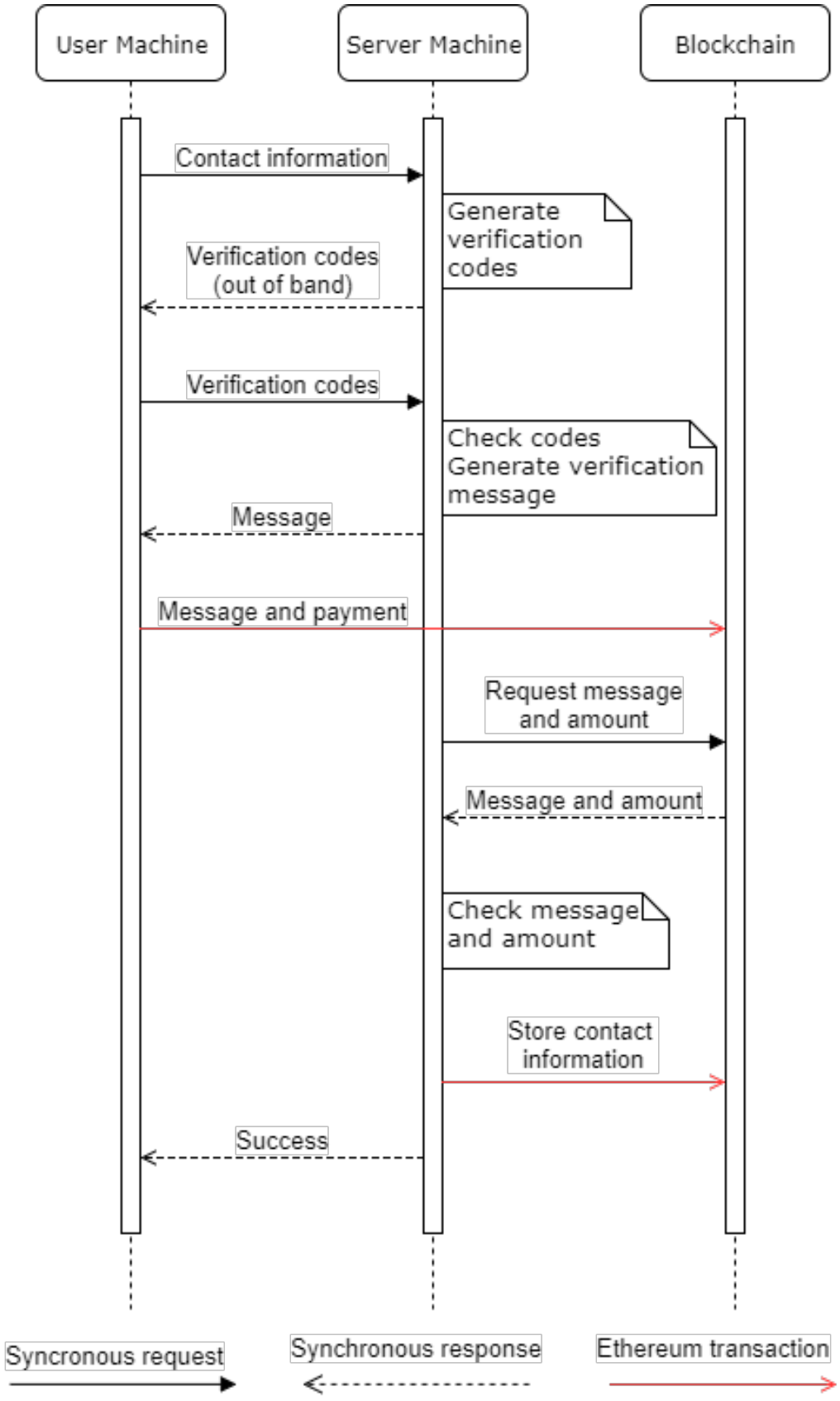

Figure 3.6: Submission interaction diagram (general)

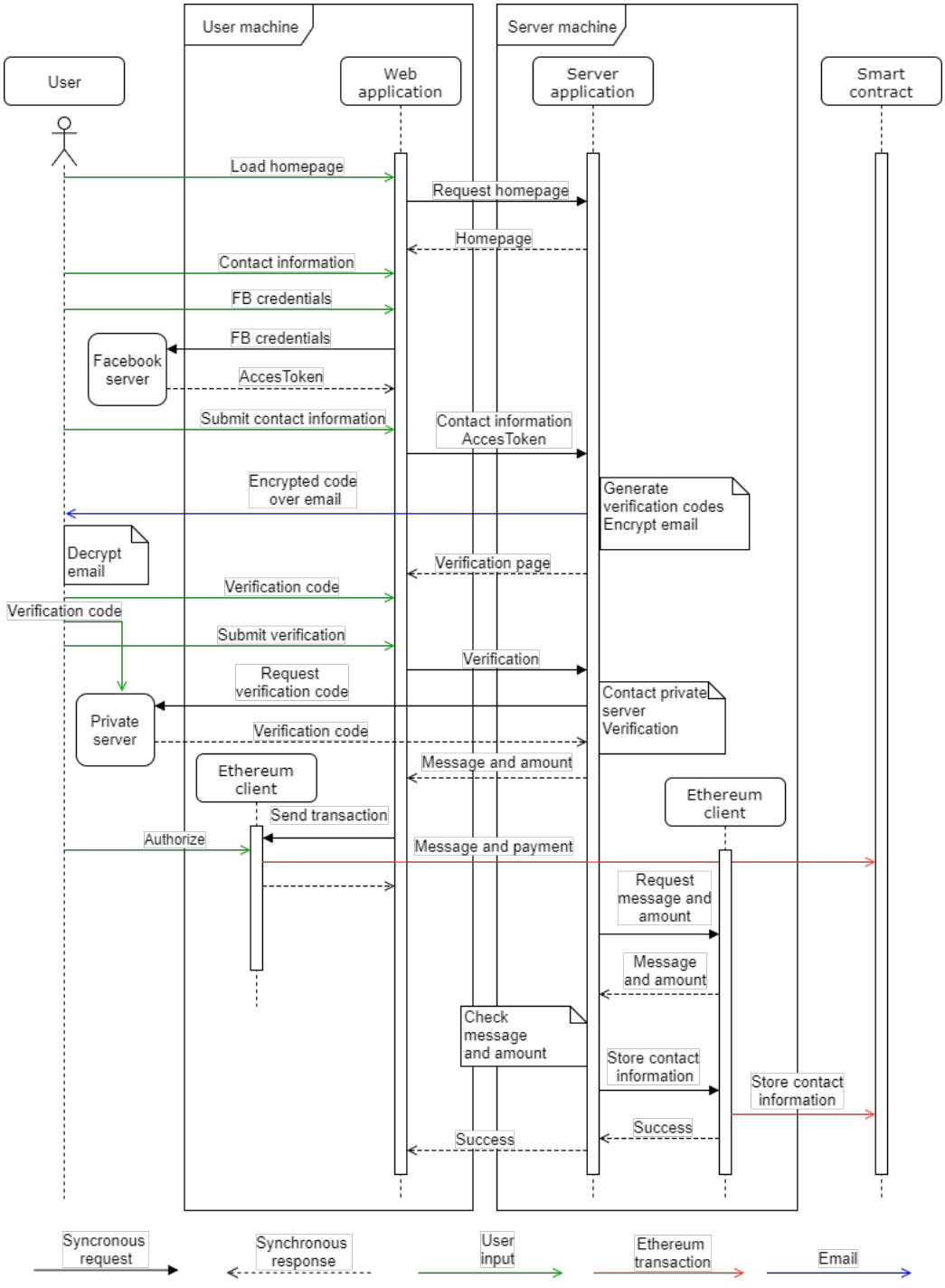

Figure 3.7: Submission interaction diagram (detailed)

we assumed that the user is able to receive the email and decrypt it using his private key.

This is the list of new entities that we introduced in figure 3.7, and for each entities we show the most important interactions involved:

- **User**: he interacts directly with the **web application**, with the **Ethereum client** holding his wallet and with the **private server** reachable at his domain address. We only represent an interaction when the user gives some input to the system, leaving out every time the user reads information from the system.

- **Web application**: it is the part of our system running on top of the browser, inside the user machine. It is the main interface for **user** interaction. The web application communicate mainly with the **server application**, but also with a **Facebook server**, to authenticate user credentials, and with the **Ethereum client (user)**, to generate the transaction that will be later authorized by the user.

- **Facebook server**:the official Facebbok server exposing the API, reachable at `https://graph.facebook.com/`.

- **Private server**: the server under the control of the user reachable at the domain address that is being verified. The verification process involves the **user** uploading a file containing the code on the server and making it public. Then the **server application** requests the file and verifies that the code matches.

- **Ethereum client (user)**: it contains the user's wallet and can send Ethereum transactions. It interacts with the **user** through a GUI and with the **web application** through the Web3 JavaScript API. It is essential to pay the storage fee and to verify the Ethereum address.

- **Server application**: the core of the system running inside the server machine. It incorporates most of the application logic. It interacts with the **web application** through HTTP requests and with the **Ethereum client (server)** through the Web3 JavaScript API. The verification email is also sent from the server application to the user, but we did not decide to model the email server as a separate entity.

- **Ethereum client (server)**: the entity that the web application must use to interact with the **smart contract**. It contains a wallet to pay for the storage fee, and also to collect users' fees from the smart contract.

- **Smart contract**: the only entity of the system running on the blockchain. It can interact with others entities only through Ethereum transactions and it is completely passive. Its role is to verify the user Ethereum address, collect fees and store the contact cards long term on the blockchain.

Look-up

The look-up use case is much simpler, compared to the submission case. It is represented at figure 3.8. The process starts when the user inserts the Ethereum address in a search bar on the homepage. Then the address string is sent to the server application. The server application then requests the contact information to its Ethereum client. The Ethereum client calls his local copy of the smart contract to check if any information exists. If that is the case, it returns the contact card to the server application. The contact information is then returned to the web application and displayed to the user.

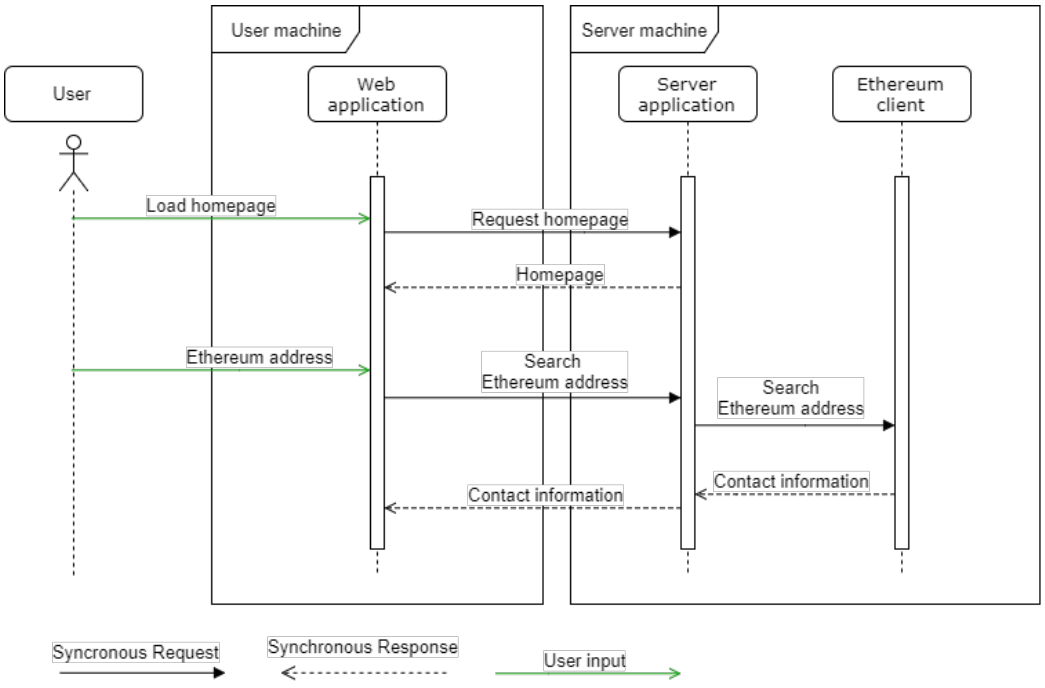

Figure 3.8: Look-up interaction diagram

Revocation

The revocation use case is represented at figure 3.9. The process starts when the user submits to the revocation page the Ethereum address that identifies the contact card that needs to be revoked. The web application generates a transaction and sends it to the Ethereum client for authorization. The user then authorizes the transaction, which is sent to the smart contract address. The smart contract, once activated by the transaction checks if there is a contact card identified by the sender address; if that is the case the contact card id marked as revoked.

The user can also revoke the contact information independently of our system, generating on his own the revocation transaction, based on the public interface of the smart contract. Then the transaction must be sent to the smart contract's address.

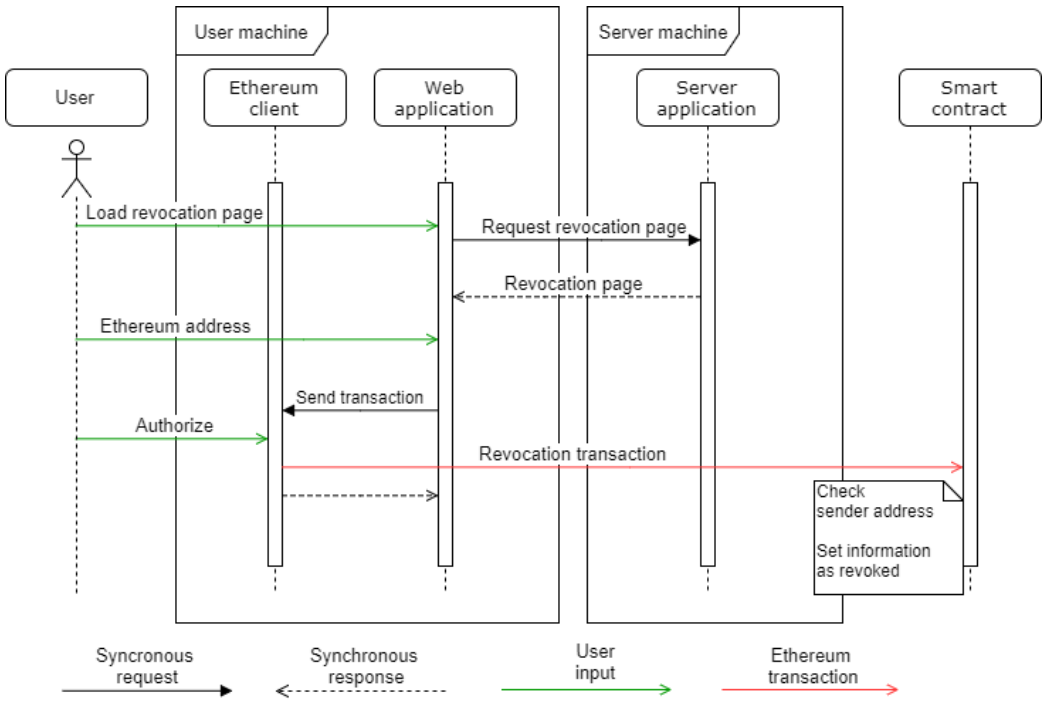

Figure 3.9: Revocation interaction diagram

3.3 Testing

Although no automated testing has been designed for the system, manual testing will be performed during the implementation phase. This will be

facilitated by the fact that the system has been designed having in mind the principle of separation of concerns. Being each part of the system responsible of certain predetermined tasks, each entity can be tested without the need to actually implement the others, but just simulating them.

Being most of the interaction based on HTTP requests and responses, we will simulate them using HTTP mock software, specifically Postman[2]. With this application we can manually define HTTP calls , send them to any URL and inspect their responses. In this way we can test server application and web application various use cases locally, before deploying them on the public Internet.

Testing in the age of blockchain

Testing a project that requires the interaction with a blockchain presents unique challenges, compared to traditional software development. It is in fact very expensive to test smart contracts and transaction on a live blockchain, because for each transaction the developer would need to pay a fee. When different functions need to be tested repeatedly to identify bugs, fees start to grow really fast.

Another challenge is that, once deployed, a smart contract cannot be easily modified. The only solution for changing the behaviour of the system would be deploying a new version of the contract to a new address and changing all the references of the old address. Doing this cannot obviously scale well beyond very simple contracts, because every piece of software needs to be maintained and modified during its life cycle to resolve vulnerabilities and bugs. It can also mine the trust in the system, because most of the appeal of smart contracts is that they are immutable.

For all these reasons, developers have come up with different ways to test Ethereum contracts without deploying them directly on the *mainnet*[3]. The most popular are the following: Ethereum client simulation, private chains and testnets.

Ethereum clients can be easily simulated using an Ethereum node simulator like *Testrpc*. Testrpc is a Node.js based Ethereum client for testing and development. When launched it can deterministically generate a locally stored blockchain, which can be used to deploy and run smart contracts, and a set of wallet containing a virtual amount of ETH. The web application and server application can then interact with Testrpc using the Web3 API like they would with a real Ethereum client. The main disadvantage of using

[2] `https://www.getpostman.com/`

[3] The mainnet is a network of nodes that share a live, functioning blockchain. This term is used in opposition to testnet, which is a network used mainly for testing.

Testrpc is that each instance of the program is local, so different entities on different machines can not interact with the same shared blockchain and smart contracts.

To solve this problem it is possible to create a private blockchain with an Ethereum client like *Geth*. A private blockchain is very similar to the mainnet, but only the nodes that are configured to connect to that specific blockchain will use it. In this way it is possible to create our own private network of nodes that share the same blockchain state. Having no competition between miners, the difficulty would be really low and we would have in a short amount of time enough ETH for any amount of testing. The drawback of this approach is that we need to have the majority of the node connected all the time to avoid splitting the chain, and also we cannot use third party online services to inspect the chain.

The last solution, and the one we adopted, is to use Geth to connect to a public testnet. A testnet is a network of nodes working on a blockchain used only for testing purpose. A testnet is usually used to test EVMs, clients, blockchain modifications and smart contracts. There are different kind of Ethereum testnet, and the most important difference between them is how the mining process works. A common feature is that getting ETH to pay for transaction fees is free or really cheap [63]. The most popular Ethereum testnets are:

- **Ropsten**: the only difference between this testnet and the mainnet is that the difficulty for mining is really low, so anyone can mine a sufficient amount of ETH for testing purposes. The problem with this solution is that attackers, trying to disrupt Ethereum development, managed to mine a lot of currency and started spamming the blockchain with cheap transactions, making it really slow[4]. Because of this the team behind the Parity client created Kovan.

- **Kovan**: the mining algorithm is not based on proof-of-work, but on proof-of-authority. In other words, new ETH are generated not by miners but by trusted validators and then distributed to users for free. This model is good to stop attackers, but this testnet is not compatible with Geth clients, only with Parity ones. Another disadvantage, but not related to our project, is that with this network is not possible for Ethereum developers to test changes involving the mining algorithm, since it does not use one.

[4] `https://ethereum.stackexchange.com/questions/12477/ropsten-testnet-is-under-kind-of-attack-what-can-we-do`

- **Rinkeby**: a proof-of-authority testnet developed by the Ethereum Foundation. Actually compatible just with Geth, it will be integrated with other clients in the future.

We decided to monitor the Ropsten for 10 days, and after detecting no notable attacks or slowdowns during such time we decided to use it for our testing. If the situation will change it will always be possible to move the smart contract to another testnet without changing the code.

3.4 Security

In a project like this, which aims at creating a service to publicly and securely store important contact information, security cannot be just an afterthought, but must be taken into account since the design phase. The most critical use case under the security point of view is the submission. We dealt with designing a verification protocol that tries to take into account many possible attack vectors. The main type of attack foreseen against our system is represented by a Man-in-the-Middle attack, where a malicious party is able to intercept every communication happening between the different entities of our system. Another attack vector would be the public facing part of the server application.

To defend ourselves from a MitM the interaction protocol, as shown at figure 3.7 had to be designed making certain assumptions about the security of the communication channels. Specifically, we assumed that:

- The HTTP interaction between the web application and server application is conducted over a secure channel, for example using a TLS protocol version that is not subject to a MitM attack. This is necessary because, even if most of the verification protocol is designed to be resistant to interception and alteration, one crucial point of information should not be known to the attacker: the Facebook AccessToken. Knowing this would allow a malicious party to control the user Facebook account. Another reason is that it must not be possible for an attacker to substitute in transit the public key submitted by the user with his own. This would allow him to intercept the content of the verification email and register a contact card with a different email address.

- The HTTP interaction between the server application and the the private server is conducted over a secure channel. This is done to prevent an attacker from registering a domain that it does not control. An

attacker could in fact intercept the request for the verification code between the server application and the private server, responding with the correct code, without controlling the specific domain.

- The PGP scheme used to encrypt the verification email is secure. Without this requirement the attacker could be able to register a public key even not controlling the relative private key. This assumption is also a foundation of the whole service, because if PGP encryption is not secure then there is no need at all for this system.

- The server application is not under the control of the attacker. Even if that condition is not respected and the attacker is able to register wrong information on the smart contract, the user would still be able see the wrong records on the blockchain, and then revoke the contact card.

- The Ethereum network works as expected and there are no attacks happening at the moment of the registration. The code of the smart contract is also assumed as immutable and executed following the EVM specifications.

- Every verification code is produced using a cryptographically secure random number generator. An attacker must not be able to guess those codes. This require also that an attacker make too many tries in the same amount of time.

With these assumption there is no need to send the verification email over a secure channel. In case of interception only the legitimate user would be able to decrypt the email containing the verification code thanks to PGP encryption.

To secure the access to the server from external attackers, we will implement the best practices against injections and cross site scripting, like input validation and sanitization. We will also have to keep the software stack running on the server updated, to avoid vulnerabilities.

It is essential for the security of the system that the smart contract behaves like required, because in case of a wrong implementation it cannot be modified easily. So we will test it very carefully before the deployment, and we will also use tools, like Securify, to identify common vulnerabilities [64].

Chapter 4

Implementation

In this chapter we will describe the process that led us to the actual implementation of the software application and its results.

The first step was deciding which tools and programming languages are best suited for the creation of each part of the system. Being already familiar with many different web technologies, deciding which tools to use for the web development part was not difficult.

Finding reliable resources about Ethereum development was much harder. We started from the official documentation[1] of the Ethereum project. Being a young platform (the Ethereum network went live on on 30 July 2015), it is considered a niche technology and it is not yet understood or know by developers not interested in cryptocurrencies. For this reason, there are not many learning resource available to the public, and the official documentation is poorly written and often outdated.

We managed to find a few online communities with good resources for starting developing Ethereum connected projects and smart contracts: on *Reddit* there are two Ethereum subreddit, for general discussion[2] and for developers[3], that are pretty active and helpful; on the *Ethereum Programming Stack Exchange*[4]; on the the *State of the Dapps* website[5], that presents a list of distributed application made with Ethereum, usually with links to the source code on *GitHub*.

The following section is about the developer tools we used to build the system, in term of programming languages, frameworks, libraries and IDE. We divided the section in two parts: one about the web development part

[1] `http://www.ethdocs.org/en/latest/`
[2] `https://www.reddit.com/r/ethereum/`
[3] `https://www.reddit.com/r/ethdev/`
[4] `https://ethereum.stackexchange.com/`
[5] `https://dapps.ethercasts.com/`

and the other about Ethereum development.

4.1 Tools

4.1.1 Web development

Client-side application

The client-side application is composed of a series of web pages generated by the server and rendered by the browser, using a mix of **HTML**, **CSS** and **JavaScript** language. We used the **Bootstrap**[6] framework as a library for the user interface.

The Facebook login authentication is handled using the official **Facebook SDK for JavaScript**[7]. Integrating the SDK on a page allows us to display a pop-up window to the user, which handles the verification of the user's Facebook credentials. Facebook requires a registration to their developers program to have access to the JavaScript SDK. After the registration we have access to an *app id* that must used together with the SDK. With the SDK we can get the *accessToken*, a string needed to obtain the user id, which is then passed to the server.

We also used **JQuery**, a popular JavaScript library for DOM traversal, manipulation, event handling and animation. We chose Google Chrome as the main browser to test and inspect the application, because of its popularity and compliance to web standards.

Server-side application

For the server application we started thinking about using a traditional LAMP (Linux, Apache, MySQL, PHP) stack as the environment running on the server machine. With this solution we would be using PHP as the server-side scripting language, while using JavaScript client-side. We though that a more homogeneous approach would reduce the time required to complete the project. So in the end we decided to change our approach, adopting the JavaScript everywhere paradigm, and we used this language both for client and server application.

After some research we considered the MEAN (MongoDB, Express, Angular.js, Node.js) stack a good solution for our needs. However we decided to not use the Angular.js framework to keep the client application simple. In the following paragraph we explain the role of each part of the stack.

[6] `http://getbootstrap.com/`

[7] `https://developers.facebook.com/docs/javascript/`

Node.js is a server-side runtime environment for executing event-driven JavaScript code. One of the main differences between an Apache and Node.js is how HTTP requests for a certain URL are handled. Apache was created in a period where static web pages were very common, so a resource is usually requested using its URL and is served as it is in a standard configuration. It is still possible to generate dynamic web pages with Apache, but for doing that a preprocessor like PHP must be enabled. Differently from other type of files, when a *.php* is requested, the code present in the file is executed by the PHP module, that dynamically generates a response, which is sent to the client, usually as standard HTML web page.

Node.js changes this paradigm, making the execution of JavaScript code as a response to HTTP requests the default behaviour. In this way creating dynamic content and interacting using AJAX calls is much simpler. Also, if we decide to use JavaScript as the client-side language, we can use the JSON as native interchange format, without any needs of wrappers or mapping to other formats.

Another big advantage of using Node.js is that is seamlessly integrated with *Npm*[8]. Npm is a package manager that helps to share and reuse JavaScript code between projects. With npm we can use a library of over 477,000 package (also called modules) directly inside Node.js, instead of having to implement many functionalities from scratch. The npm registry also handles versions and dependencies automatically. A drawback of using npm as a development tool is that there is no vetting process for submission, meaning that pakages found there can be low quality, insecure, or malicious. Another problem is that packages that add even simple functionalities can have a great amount of dependencies, making projects larger in size, increasing the computation overhead and the possibilities of vulnerabilities. For this reason it is important to select packages carefully, giving preference to the most widely used, that are also the most tested.

Express, available as a Npm package, is the de-facto server framework for Node.js. It is designed to handle HTTP request in a event-driven, non-blocking approach. With Express any application is designed as a series of routes. Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URL (or path) and a specific HTTP request method (GET, POST, and so on). Each route can have one or more handler functions, which are executed when the route is matched. Express is relatively minimal, with many features available as Npm modules.

The last piece of the stack, **MongoDB**, is a document-oriented database

[8] `https://www.npmjs.com/`

system. The reason for using a database comes from the necessity to temporary store contact information during the submission, before the verification has been completed. MongoDB is part of a growing category of non-relational DBMS, called NoSQL such databases do not support queries through SQL. While relational databases generally store data in separate tables that are defined by the programmer, and a single object may be spread across several tables, document-oriented databases store all information for a given object in a single document, and each stored object can be different from another. This makes mapping objects into the database a simple task, normally eliminating anything similar to an object-relational mapping. Document-oriented DBMS attractive for use inside web applications, thanks to their light footprint on resources and the ability to easily update the structure of documents.

Another advantage of MongoDB is that documents are stored as JSON objects. In this way we can use JSON (native to JavaScript) both as an interchange format between client and server and for storage, eliminating the need for complex mappings. MongoDB can be integrated easily with Node.js, thanks to the *mongodb* Npm package. The module installs the MongoDB Node.js Driver, that can be used to read, modify and delete stored documents through query filters. MongoDB is available in different configurations: the Community Server, that is the free open-source version, and the Enterprise Server, the commercial option that requires a license. We chose a third option, that is not installing MongoDB on our machine to relying on a database-as-a-service provider, mLab[9]. We decided to use a cloud solution because we had problems with the installation of the Community Server edition on our machine, but for the final application the database should be installed on same machine on which Node.js is running, for confidentiality and security reasons.

The stack, with the exception of MongoDB, was installed on a general purpose local machine for testing, with the possibility of migrating it to a single purpose server in the future. Other than the components that previously described, we installed also the following npm packages on Node.js:

- *openpgp*[10]: OpenPGP.js is a JavaScript implementation of the OpenPGP protocol. It is used as a library to handle the armored public key and to encrypt the verification email that is sent to the user. We also used it outside Node.js, in the web application that runs on the user machine: in this context is employed for the formal validation of the armored public key and for extracting the user's email, if present.

[9] `https://mlab.com/`

[10] `https://github.com/openpgpjs/openpgpjs`

- *nodemailer*[11]: a module that can securely connect to SMTP server over the TLS protocol and send emails. It is used for sending the verification email trough a dummy Gmail account we created for testing.

- *request*[12]: a simple module to make HTTP requests from the server. It is used to verify the user's domain address.

- *body-parser*[13]: a Node.js HTTP body parsing middleware. It is used to parse and extract the content of the HTTP requests received by Node.js.

- *express-mongo-sanitize*[14]: a middleware to sanitize the content of HTTP request. This is done so that way we can pass the parameters directly to MongoDB queries, but without the risk of *query selector injection* attacks [65].

- *jade*[15]: Jade, also known ad Pug, is template engine that can generate HTML code. We used it because Jade syntax is cleaner and without many of the complexities of HTML, so that we don't need to close tags and write a lot of parenthesis. For example, the following Jade code:

```
.topic-component
    h4 Title
    p Description
    a(href="#") Link
```

is converted to HTML as:

```
<div class="topic-component">
  <h4>Title</h4>
  <p>Description</p>
  <a href="#">Link</a>
</div>
```

- *web3*[16]: this is the Ethereum JavaScript API, which is required to interact with a local or remote Ethereum node. The module communicates

[11] `https://nodemailer.com/about/`
[12] https://www.npmjs.com/package/request
[13] `https://github.com/expressjs/body-parser`
[14] `https://www.npmjs.com/package/express-mongo-sanitize`
[15] `https://pugjs.org/api/getting-started.html`
[16] `https://github.com/ethereum/web3.js/`

with the node using JSON-RPC (a stateless, light-weight remote procedure call protocol). This is required for Node.js to interact with the Ethereum client running on the server machine.

Integrated development environment

Since many technologies are shared between the server-side application and the client-side application, we decided to use an IDE that could support the development of both. Our choice was WebStorm from JetBrains[17]. This IDE includes an editor with syntax highlighting and autocompletion for many different languages, including JavaScript, HTML, Pug and Solidity. Integration with Node.js and Npm is also very intuitive, both for debug and deployment. It is possible to run a Node.js instance directly from the IDE and to interact with it through a normal web browser.

We decided to not use a *version control software* like Git, SVN, CVS, etc. Since we do not have a team of multiple developers working on the same project at the same time, we decided to avoid versioning altogether. In hindsight we realized that we could have avoided many problems, most of them involving keeping track of changes, using a version control software.

4.1.2 Ethereum development

The first decision we had to make regarding the development of the Ethereum part of the project was choosing the smart contract's programming language. We decided to implement the smart contract using Solidity. The main reason is that Stolidity is the most popular high-level language for contracts, and so it has been tested and has the most detailed documentation (even if is still poor compared to general purpose programming languages) [38]. Another reason is that, while being contract-oriented, it has a syntax similar to JavaScript, that we used almost exclusively for the web part.

WebStorm can support Solidity through an official plugin, but we decided to use the Remix as IDE, since it was specifically created for Ethereum smart contracts. Remix is browser-based and offers a Solidity integrated compiler and a runtime environment without server-side components. It can be used to deploy smart contracts. The deployment can be done in a virtual runtime, which allows the test of simple methods that do not require interaction with other contracts, or it can be connected to a Web3 compatible Ethereum client, local or remote. Remix includes a debugger for both Solidity and EVM compiled code and can do an estimate of the amount of gas needed to execute a transaction call.

[17] `https://www.jetbrains.com/webstorm/`

We chose Geth as our Ethereum node running on the server machine. We used it also as the user's Ethereum wallet, in conjuntion with Mist. Geth is the most popular client implementation and is written using Go language. It can be used through a command line interface and the Web3 JavaScript interface. With Geth we can do everything needed to interact with the Ethereum blockchain and smart contracts: sending transaction to move funds between wallets; calling smart contact's procedures; deploying contracts; exploring block history; mining ETH; managing accounts.

Because testing on the Ethereum mainnet require spending ETHs, which are quite expensive, at first we tested our smart contract using the Remix internal runtime. Then to evaluate the integration between the Node.js server and the Web3 API we used the Testrpc tool to simulate the behaviour of a real node, as explained in section 3.3. Finally we deployed the smart contract on the Ropsten testnet. As default behaviour, Geth connects to the mainnet when is launched; to use the Ropsten testnet it should be launched with the parameter *–testnet*.

Etherscan is another web tool that we used for Ethereum development. It is a blockchain explorer, so it can be used to see details about the blockchain, like addresses, transactions, blocks and tokens, both on the mainnet and on various testnets. A very useful feature is that it can also show the code of a specific contract, and can be used to interact with it. Calling a contract's procedure through Etherscan is possible only if it does not contain instructions to write data on the blockchain: if that is the case, a transaction must be used, since is the only way to pay the data storage fee. Even with this limitation it can be very useful, for example to check if the contact card has been stored correctly on the contract after the submission use case.

Since contracts are stored on the blockchain as compiled ASM code, additional steps are required after the deployment so that Ethscan can recognize the contract's interface (procedure calls, variables, structures, etc.). The contract must be verified by uploading to Etherscan its Solidity code and the ABI (Application Binary Interface). Etherscan then compiles the Solidity code into ASM and compares it to the code stored blockchain; if both match it publishes the Solidity code on the contract's page for everyone to see. The ABI is a standard that describes the list of contract functions and arguments, so that Etherscan can know how to encode parameters passed to the contract and how to decode return values. We can consider an ABI like a low-level API.

As we said before, the user, during the submission phase and the revocation phase, must send Ethereum transaction to the smart contract. One way of doing that is using a Ethereum-aware browser, like Mist, or using a

Chrome plugin, like Metamsk[18], that can receive transaction requests from the web application through Web3. We decided to install Mist and set it on the Ropsten network to test both use cases.

The last tool that we used for smart contract's development is Securify. It is a online service that can scan the contract code for common security vulnerabilities. Some of them, given the nature of the EVM, are not trivial at all. As an example, a contract is exposed to a *transaction reordering vulnerability* if a miner (who executes and validates transactions) can reorder the transactions within a block in a way that affects the amount of ETH sent to the receiver. So it is always recommended to check for common vulnerabilities with Securify or other similar tools.

4.1.3 Other tools

These are other various tools that we used for testing and development:

- **Postman**: application for sending HTTP requests, saving responses, and automate interaction. We used it to test the web server without the need interact with the web application.
- **Gmail**: free email service. We created two dummy email accounts: one to serve as the user address that must be verified, and the other for the server to send verification emails.
- **CryptUp**[19]: PGP encryption tool. It is a Google Chrome plugin to decrypt and compose encrypted email through Gmail. It was used to generate PGP key pairs used for testing and to read the encrypted emails sent for verification.
- **Altervista**: free web hosting service with customizable subdomains. We registered a free website reachable at `provawoo.altervista.net` This was needed to simulate the user's domain verification process.

4.2 Application

After defining which tools and language to use, we wrote the actual code of the application, following the design that we came to after analyzing the requirements. The application is divided between the Node.js server (JavaScript, HTML, CSS, Json) and the smart contract (Solidity).

[18] `https://metamask.io/`
[19] `https://cryptup.org/`

4.2.1 Configuration

Since the server must interact with different components, having them hard-coded on the software would be not practical. For this reason we introduced a JSON file to store the configuration parameters, that is loaded every time the server is started. An example of the configuration file is showed at figure 4.1. The parameters included are:

- *environment*: *dev* if the contract is deployed on testnet, or to *live* for using the mainnet.
- *mongodbConnectionString*: connection information for the MongoDB database. It includes username, password and the URL that points to the running instance.
- *email*: username, password and URL of the SMTP server.
- *smartContract*: smart contract's configuration parameters:
  - *bin*: contract's binary code (not used).
  - *ABI*: contract's interface in ABI format.
  - *wallet*: public address of the wallet controlled by the server, on testnet and mainnet.
  - *contractAddress*: contract's public address, on testnet and mainnet.
  - *rpc*: address and port of the Ethereum node, on testnet and mainnet.

4.2.2 Server interface

The interaction between the web application, running on the browser, and the server application is carried over HTTP requests. We can consider such requests as methods identified by the relative path of the URL. For example the method *submit* is called when the user send a request for the resource located at address `*userdomain.tld*/submit`.

The following scheme defines the arguments and expected behaviour of each method.

- ***/signup***
  - **arguments**: none.

Figure 4.1: Configuration file

  - **description**: return the submission page (homepage).

- ***/submit***

  - **arguments**: *address, email, pubkey, facebook, domain.*
  - **description**: generate random email verification code; generate random domain verification code; save verification codes and contact information on MongoDB; return verification page.

- ***/verifyDomain***

  - **arguments**: *address.*
  - **description**: return true and set as verified on MongoDB if the domain verification is successful, false otherwise. The verification is done comparing the code saved on MongoDB and the code saved inside a JSON file saved at `*userdomain.tld*/domainCode.json`

- ***/verifyEmail***

  - **arguments**: *address, emailCode.*

  - **description**: return true and set as verified on MongoDB if the email verification is successful, false otherwise. The verification is successful if the submitted *emailCode* matches the the code saved on MongoDB.

- ***/getMessage***
  - **arguments**: *address*.
  - **description**: if both email and domain are verified, return the verification message. The verification message is created as the hash (SHA256) of the contact information, including verification codes.

- ***/checkTransaction***
  - **arguments**: *address*.
  - **description**: check if the Ethereum transaction (verification and payment) has been sent to the smart contract. If the message sent with the transaction matches the one saved on MongoDB and the payment is over a certain value, return true, otherwise return false.

- ***/search***
  - **arguments**: *address*.
  - **description**: if the address has been registered on the smart contract, return a web page displaying the contact information, otherwise return a *noResult* page.

- ***/revocation***
  - **arguments**: *address*.
  - **description**: return the revocation page.

4.2.3 Smart contract interface

We created a smart contract, called *storageContract.sol*, written in Solidity. A contact card, as defined in the contract, is a structure identified by an Ethereum address with public fields listed in table 4.1.

Server and users can interact with the smart contract through public procedures. The following scheme defines the arguments and expected behaviour of each procedure.

- **(default procedure)**

Type	**Name**	**Destripction**
string	email	Email address.
string	pubKey	Armored public key.
string	domain	Domain address.
string	facebook	Facebook identifier.
uint	payment	ETH amount paid by the user.
bytes	hashData	Ethereum address verification message.
boolean	revoked	True if the contact card is revoked.

Table 4.1: Contact card

  - **description**: called every time the contract receives a transaction without any procedure specified. Create a new contact card identified by the sender's address and set the *payment* field as the amount of ETH attached to the transaction and set the *hashData* field as the message attached to the transaction.

- **newIdentificationData(address *addr*, string *email*, string *pubKey*, string *facebook*, string *domain*)**

  - **description**: store the arguments in the contact card identified by the Ethereum address (*addr* argument). Callable only by contract's owner (server).

- **sendEtherToOwner()**

  - **description**: send all ETH paid to the contract to the owner of the contract. Callable only by contract's owner (server).

- **revoke()**

  - **description**: set the contact card identified by the sender's Ethereum address, if present, as revoked.

- **kill()**

  - **description**: delete every contact card and delete the smart contract. Callable only by contract's owner (server).

For each contact field, there is also a procedure that, given the Ethereum address of the contact card, returns the content of the field.

4.2.4 Use cases

In this section we will describe how the application behaves during the different use cases, both from the user's point of view (user interface and interaction) and from the point of view of the components of the system.

Submission

This is the most complex use case. The user first loads the application's homepage (figure 4.2). Then proceeds to fill each field of the form with his contact information and public key. If the armored public key contains an email address, the user can extract and copy it directly in the email field, pressing the "Get email" button. This is possible thanks to the OpenPGP.js library, that is executed client-side to check if the public key is formally valid and to extract the email.

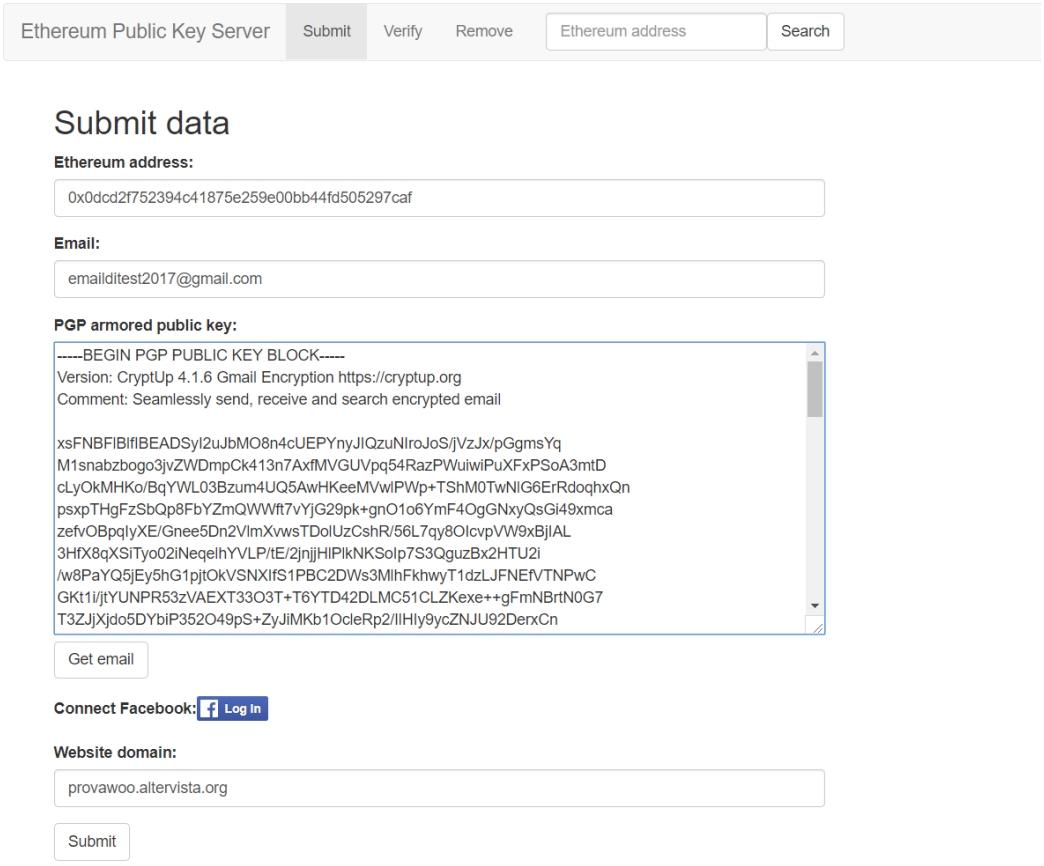

Figure 4.2: Homepage

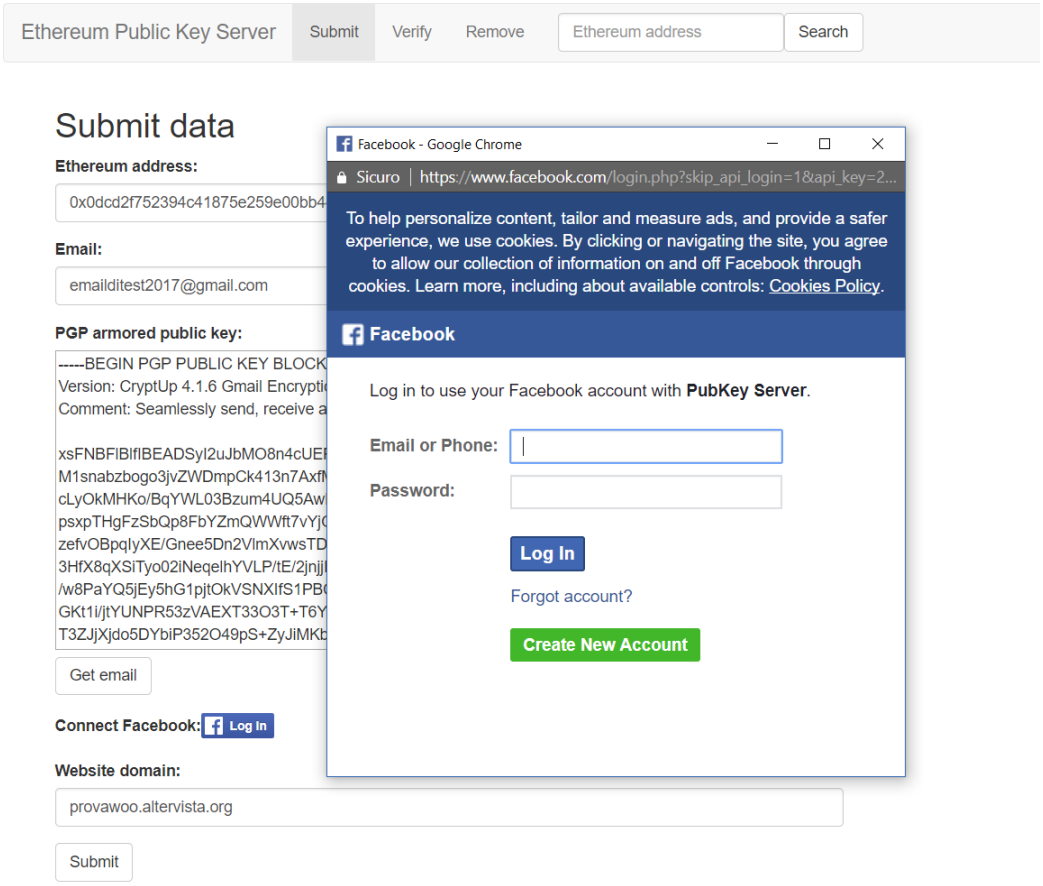

Figure 4.3: Facebook login

Then the user clicks on the "FB Login" button, that launches a pop-up authentication window, as shown at figure 4.3. Once he has authenticated with his Facebook credentials the window disappears. Under the hood the web page retrieves the user's *accessToken* using the Facebook JavaScript SDK.

Once all the information have been entered, the user clicks on the "Submit" button. A validity control on each field is executed on the web page. If the validation is not successful for all fields, a red warning is displayed near the invalid input. If the validation is successful, the web application calls the **/submit** method, sending to the server the Ethereum address, email address, public key, *accessToken* and domain.

The server application executes the */submit* method through the following steps:

- Two random verification codes (*emailCode* and *domainCode*) are generated using the *crypto*[20] module. They are 8 bytes long and encoded as hexadecimal strings.

- The Facebook unique identifier is retrieved sending a GET request to the Facebook Graph API. The request is created follwing this scheme: `https://graph.facebook.com/me?fields=id,name&access_token=` + *accessToken*. The *id* is then returned to the server.

- Contact information and verification codes are saved on MongoDB, with a time-to-live of 24 hours, so that the user can resume a verification process that has been interrupted. At figure 4.4 is represented an example of a MongoDB document storing a contact card.


```
{
  "_id": {
    "$oid": "5966281cdf4d61024057ec48"
  },
  "address": "0x0dcd2f752394c41875e259e00bb44fd505297cab",
  "email": {
    "value": "emailditest2017@gmail.com",
    "verificationString": "bfbaf653ce20187a",
    "verified": false
  },
  "pubKey": "-----BEGIN PGP PUBLIC KEY BLOCK-----\r\nVersion: CryptUp 4.1.6 Gmail Encryption
  "facebook": {
    "id": "10212673381939852",
    "name": "Pier Francesco Costa"
  },
  "domain": {
    "value": "provawoo.altervista.org",
    "verificationString": "bb6867c10c1ceb4b",
    "verified": false
  }
}
```


Figure 4.4: Contact card stored on MongoDB

- An an email containing the verification code is sent to the user's address. The bosy of the email is encrupted through OpenPGP.js using the user's private key.

[20]The *crypto* module provides cryptographic functionality that includes a set of wrappers for OpenSSL's hash, HMAC, cipher, decipher, sign and verify functions. OpenSSL is a robust, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. It is also a general-purpose cryptography library. Given its extensive adoption we consider the underlying OpenSSL implementation secure, assuming that the Node.js run-time is updated to the latest version [66]

- Finally, the verification page is returned to the user's browser.

The verification page is represented at figure 4.5. At this point the user must decrypt the email received and enter the verification code in the "verification code" form. Then, when the user press the "Verify email" button, the web page calls the */verifyEmail* method, passing the email address as the argument. If the verification is successful, the email is set as verified in the MongoDB document and the method returns successfully. Then the user must click on the "Download" button to download the `domainCode.json` file and upload it on his own server at the address `*userdomain.tld*/domainCode.json`. After uploading the file he clicks on the "Verify domain" button, that calls the server method */verifyDomain*. If the method is successful the domain is set as verified in the MongoDB document and the method returns successfully. .

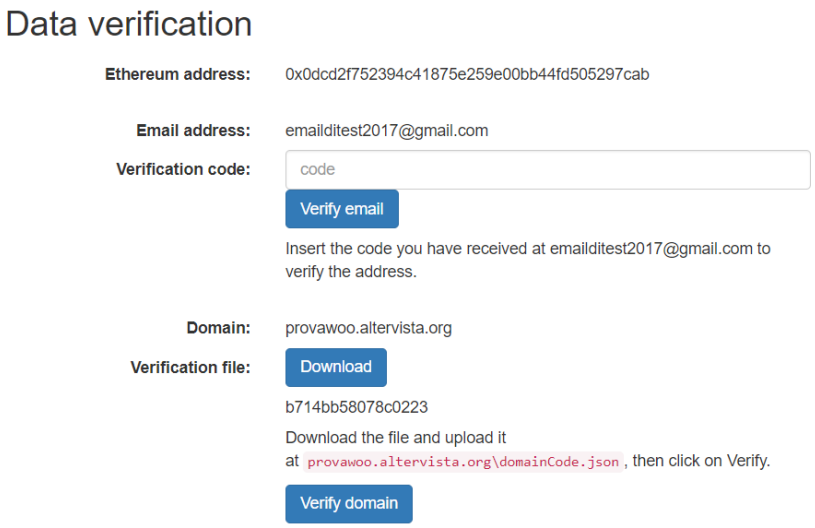

Figure 4.5: Verification page

When both email and domain are verified, the web page displays the smart contract address, the verification message and the amount of ETH to pay. The verification message is generated as the hash of the MongoDB document of the contact card. The hash is computed using the SHA256 algorithm offered by the *crypto* Node.js library. Since the document includes two random codes, then also the hahs of the document is assumed to be random.

The user then needs to send a transaction to the smart contract from the

Ethereum address the he wants to verify, with the right amount of ETH[21]. If the Mist browser is being used, then the user can just click on the "Send transaction" button. This would send a transaction request (with the correct amount and message) to Mist, that would prompt to the user to authorize it with the wallet password, through a pop-up window (figure 4.6). Otherwise the transaction could be sent manually.

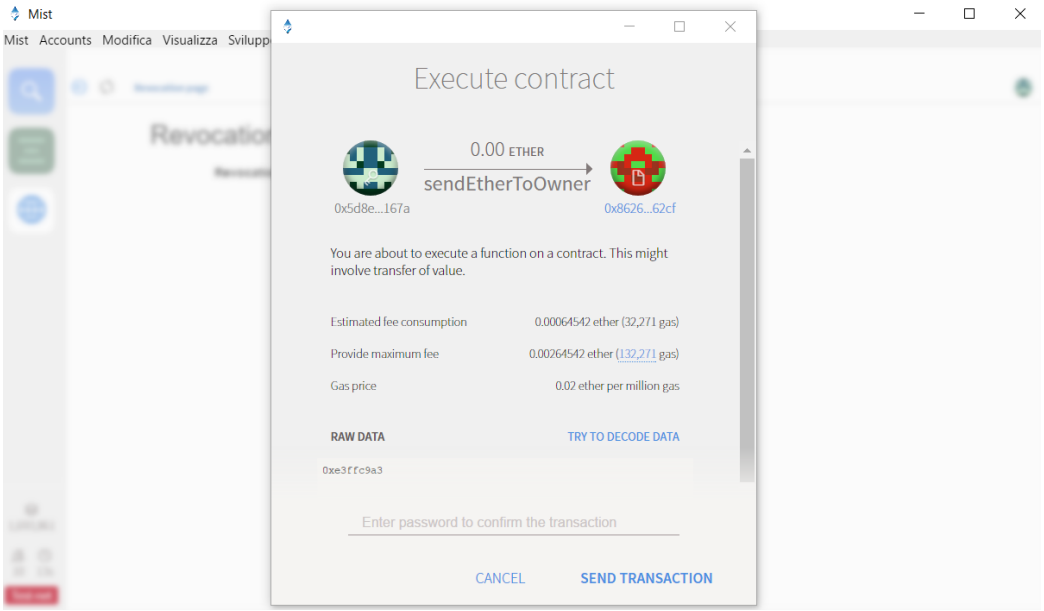

Figure 4.6: Transaction authorization (Mist)

Once the transaction has been included in a block, the user can press the "Check transaction" button: the web page calls the */checkTransaction* server method, that verifies the correctness of the message and the amount. The server then proceeds to store the contact information calling the smart contract procedure *newIdentificationData* and paying the relative fee[22]. Finally a *Success* web page is displayed to the user, that can then search his Ethereum address to see the contact information as stored on the contract.

[21] The analysis on the right amount of ETH to pay for the storage cost will be discussed in the Conclusions chapter

[22] The server owner must from time to time, call the *sendEtherToOwner()* smart contract procedure. In this way he can withdraw the ETH fees paid by users, and recover the cost sustained for on-chain storage

Look-up

The look-up use case is simple. The user can search an Ethereum address through the search bar present on the top of homepage. The request is sent to the server, that returns a no result page if the address is not registered on the smart contract. Otherwise it returns a web page showing the contact card with all the information for that address (figure 4.7). The server queries his local copy of the blockchain through the Web3 API, as a read-only procedure, so no fee is required.

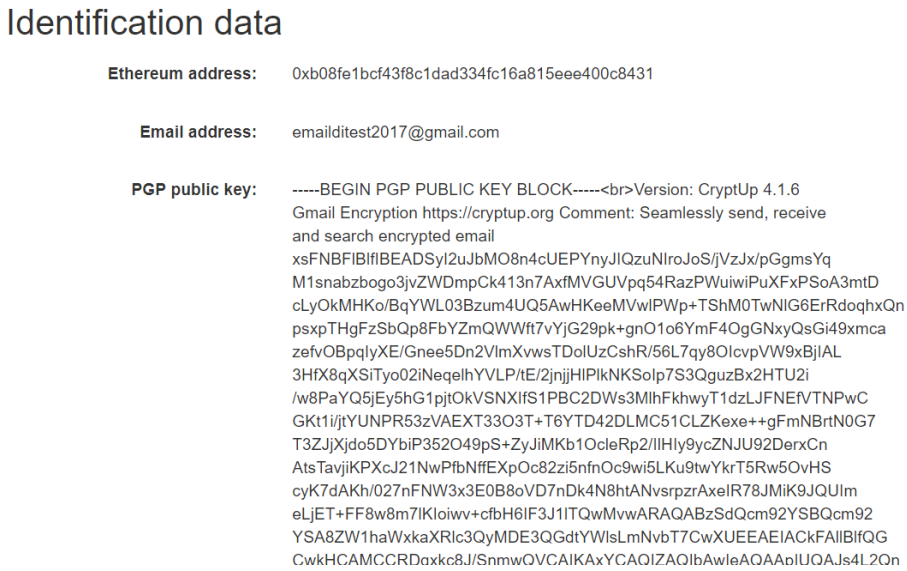

Figure 4.7: Contact information look-up

Revocation

A contact card stored on the contract can be revoked only if the owner of the Ethereum address sends a transaction call to the *revoke()* procedure. The procedure simply sets the *revoked* flag of the contact card to true. We decided to not delete completely the contact card, so an attacker that has taken control of another person's address, can not upload a fake poisoned contact card with the same address.

The transaction can be sent with Mist, though the official revocation page (figure 4.8), or as a manual transaction from any Ethereum wallet. The only requirement is that the sender's address must be the same one of the contact

card. A the fee for the execution will be included in the gas paid by the user to send a transaction.

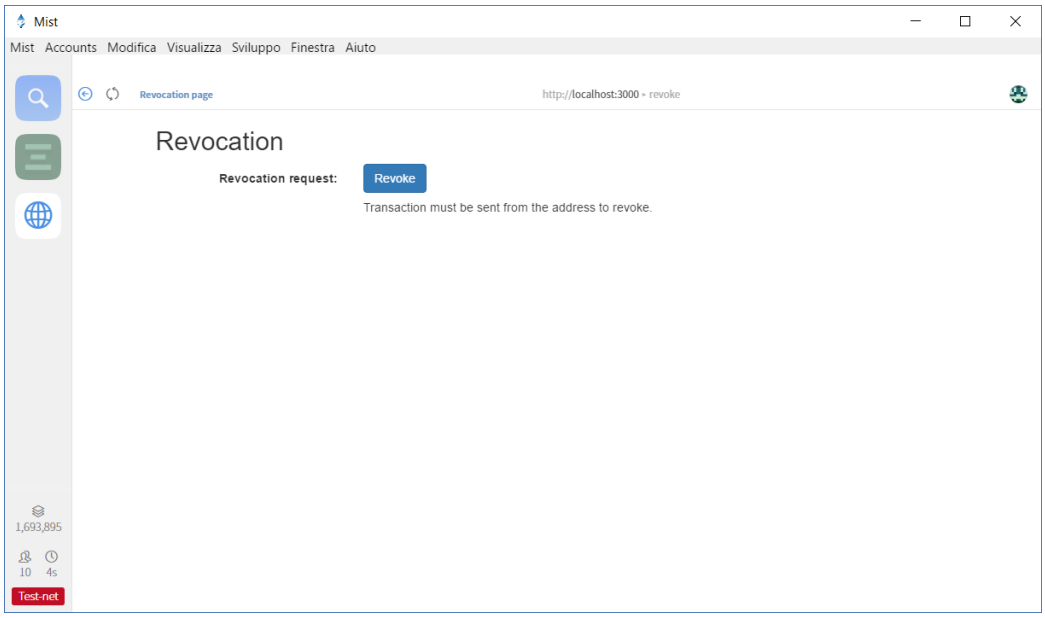

Figure 4.8: Revocation page

Chapter 5

Conclusions

5.1 Application

Coming to the conclusions, we must consider the result of the thesis. We built a prototype of the application that we proposed in the introduction chapter: a decentralized and autonomous key server. The application has been tested for the three use cases that we defined: submission, look-up and revocation. The system allows users to verify and store their PGP public keys and other contact information(Ethereum address, email address, domain address, Facebook identifier) on the public Ethereum blockchain instead of relying on a traditional key server.

We do not think that this is the final solution to the problem of public key discovery for secure communication, but we feel that our application can be a useful tool in addition to the Web-of-trust and traditional key servers. In computer security field is always better to reduce the amount of trust users need to put in third party services, and with the proposed application we make a step in that direction. We could have realized a more autonomous service if managed to implement also on-chain verification, but for technological and economic reasons explained in the design chapter, we decided to opt for off-chain verification.

We must note that the smart contract was not deployed on the mainnet, but on the Ropsten testnet and the server application was tested on a local machine and not over the Internet. Regardless of this situation it is our opinion that the deployment of the application for public use would be almost painless. The remaining steps are:

- Deploying the contract on the Ethereum mainnet. The costs will be discussed the next section.

- Buying or renting a dedicated server and installing the server application on top of it.
- Registering a domain address for the server.
- Acquiring a X.509 certificate from a trusted Certificate Authority to enable strict TLS encryption for the server.
- Changing the configuration file of the server application to live mode, and adding the address of the mainnet contract.

The service manager should also set the ETH fee that must be paid by the user to upload his contact card. The cost aspect is discussed in the following section.

5.2 Cost analysis

In this section we will analyze the costs required to run an application that integrates a smart contract. We are not interested in the standard operational costs of a web application (hosting/housing, domain registration and renewal, TLS certificate, etc.), but on the specific Ethereum costs required to execute code on the blockchain, and how to split them between the user and the owner of the server.

As we said previously, each computation on the blockchain consumes a certain amount of *gas*. Such gas must be provided by the transaction call that set off that particular execution. Specifically, in each transaction, a *gasLimit* and a *gasPrice* are set by the sender. $gasLimit \times gasPrice$ is the maximum amount of ETH that the sender is ready to spend for the transaction. The amount of ETH that is actually paid in the end is equal to the $gasUsed \times gasPrice$.

At first, a fee must be paid by the owner to deploy the smart contract on the blockchain. We can compute the cost looking at the *gas* that we spent to deploy the contract on Ropsten, since the *gas* spent for a transaction on the testnet is the same that would be spent on the mainnet. The only difference is that on Ropsten is much easier to generate ETH. Since the amount of *gasUsed* was 2043181 at and the *gasPrice* was 0.00000002 ETH, we would have spent 0.04086362 ETH. A the current price of 300 $ per ETH (September 2017) that would be **12 $**. This price is almost exclusively represented by the size required to store the compiled code, so it can be decrease reducing the number of instructions. The contract, written in Solidity, is 98 lines of code.

The second cost that we must consider is for calling the **newIdentificationData()** procedure. This is paid also by the server. Since this is a recurring cost, we need to make the user pay for it. We can do that setting the same amount as the fee paid during the verification transaction. The cost of the procedure is 0.04746934 ETH $2373467 gasUsed \times 0.00000002 (gasPrice)$ or 14 $. We must note that this cost is heavily influenced by the size of the PGP key, because it is the biggest variable that needs to be stored on the blockchain. 14 $ is required when using a 4096 bit PGP key, while it became 9 $ using a smaller 2048 bit key.

After this calculation we decided for a 0.05 ETH fee that must be paid by the user with each verification transaction, plus transcation fees. We also introduced a size check for each parameter, so that users can not abuse the system with very long parameters, forcing the server to send very expensive transactions. There is nonetheless a problem with a static fee: if the *gasPrice* will change in the future it would be impossible to change the fee. An improvement could be having a dynamic fee on the smart contract that can be changed through a transaction call from the server.

Finally we need to evaluate the costs required to call the **revoke()** procedure and the **sendEtherToOwner()**. The revocation is going to cost to the user a little more than a normal Ethereum transaction, since changing the value of a variable does not require any storage space, since the memory has already been allocated. The gas needed by the **sendEtherToOwner()** procedure is around two times a normal transaction, since it just sends back another transaction. So the cost of the two procedures are, respectively, 0.00042 ETH (0.12 $) and 0.00084 ETH (0.24 $).

We conclude that the cost, even if not affordable to store many contact cards per user, make sense for this kind of service, since the data is stored forever on the blockchain, while the fee must be paid only once. Anyway, we have to take into consideration that the ETH price is extremely volatile and it so costs could change in the future.

5.3 Limitations and future work

While this thesis has demonstrated the viability of a decentralized and autonomous key server, many opportunities for extending the scope of this thesis remain. This section presents some of these directions:

- **On-chain verification.** Further research would be needed to create a completely trustless application, that relies only on smart contracts for both storage and verification. A similar solution would greatly increase the utility, security and availability of the application.

- **More social network accounts and optional fields.** The implemented solution can store just a Facebook account, and it is mandatory. Since not everyone has a Facebook account, it could be useful to widen the compatibility with different social network accounts. Making every field of a contact card optional, with the exception of the Ethereum address and the PGP public key, would also greatly improve the flexibility of the service. Then it would be necessary to find a way to make the user's fee dynamic for each contact card registration, since the size would vary.

- **Cost reduction.** We recognize that the application is quite expensive at the moment. Since data storage is the biggest cost for the application, the only way of reducing it would require a smaller space allocated to the contact card, specifically to the public key. A way of doing that could be saving only the fingerprint of the key and then storing the complete key in some other system, may it be distributed like IPFS or Swarm, or centralized, like a normal key server.

- **Adjustable fee.** As we previously highlighted, it could be useful to find a way to change the user's fee. One way could be using a non-static variable inside the contract and changing that through a procedure called by the server. Another way could be making the contract aware of the *gasPrice*, so that it could set the fee autonomously.

- **Update contract.** One of the biggest disadvantages of smart contract is that they can not be updated if no explicit mechanism has been included by the creator. One of the drawbacks of our application is that, if the contract must be changed for any reason, the only way of doing that would be deleting it and creating a new one, losing all the old contact cards. It would be interesting to find a way of updating the contract, but still keeping the system trustless.

5.4 Ethereum development considerations

The work done for the thesis has gave us a better understanding of what it means to build application with blockchains, and specifically designing smart contracts on the Ethereum platform. We came to the conclusion that the technology is still in its infancy and not mature enough for mainstream use, but it is rapidly getting better and more capable. One of the biggest disadvantage is the lack of resource and reliable documentation for developers, but the situation is improving thanks to the growing popularity. Tools

that greatly simplify designing, testing and deploying contracts are being developed.

We recognize that the blockchain is a revolutionary technology, but is still difficult to find use cases outside crypto-finance and notarization of public information. We think that the reason is that we cannot make a blockchain interact with the real world without losing its trustless nature, since interacting with external entities requires third party bridges. But being trustless is the only reason to prefer a blockchain to some other solution, given its scalability and performance limits.

Another drawback that we did not explore is the incredibly inefficient proof-of-work model, that wastes a vast amount of energy for no additional gain, other than generating value for miners. In an era in which almost every industry is going in the direction of being eco-friendly and more sustainable, it seems illogical to adopt a technology that literally burns energy. A solution to this question will hopefully arrive, at least for Ethereum, with the scheduled switch from proof-of-work to proof-of-stake.

Despite this doubts, the cryptocurrency community is so full of innovation and optimism that is difficult to consider the technology just a bubble or a fad. If the ups and downs of Bitcoin over the last 9 year have though us something, is that blockchains are here to stay, and we will have to deal with them for a long time.

Bibliography

[1] Oxford Dictionaries. *Cypherpunk*. URL: https://en.oxforddictionaries.com/definition/cypherpunk (visited on 04/21/2017).

[2] David Chaum. "Blind signatures for untraceable payments". In: *Advances in cryptology*. Springer. 1983, pp. 199–203. URL: http://blog.koehntopp.de/uploads/Chaum.BlindSigForPayment.1982.PDF (visited on 03/24/2014).

[3] David Chaum and Stefan Brands. "Minting'electronic cash". In: *Spectrum, IEEE* 34.2 (1997), pp. 30–34. URL: http://homepage.cs.uiowa.edu/~cremer/courses/cs2/ecasharticle.pdf (visited on 03/24/2014).

[4] Wei Dai. *B-money*. 1998. URL: http://www.weidai.com/bmoney.txt (visited on 03/24/2014).

[5] Adam Back et al. *Hashcash - A Denial of Service Counter-Measure*. 2002. URL: ftp://sunsite.icm.edu.pl/site/replay.old/programs/hashcash/hashcash.pdf (visited on 03/24/2014).

[6] Hal Finney. *RPOW*. 2004. URL: https://web.archive.org/web/20071222072154/http://rpow.net/ (visited on 03/25/2017).

[7] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: https://bitcoin.org/bitcoin.pdf/ (visited on 04/27/2014).

[8] N Gregory Mankiw. *Principles of macroeconomics*. Cengage Learning, 2014.

[9] *Price Chart History*. URL: https://99bitcoins.com/price-chart-history/ (visited on 04/21/2017).

[10] *Bitcoin Avg. Transaction Fee Chart*. URL: https://bitinfocharts.com/comparison/bitcoin-transactionfees.html (visited on 07/21/2017).

[11] *Bitcoin scalability problem*. URL: https://en.wikipedia.org/wiki/Bitcoin_scalability_problem (visited on 07/21/2017).

[12] John Bohannon. *Why criminals can't hide behind Bitcoin.* Science Mag. URL: http://www.sciencemag.org/news/2016/03/why-criminals-cant-hide-behind-bitcoin (visited on 04/21/2017).

[13] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C.* john wiley & sons, 2007.

[14] Jan Leeuwen. *Handbook of theoretical computer science.* Vol. 1. Elsevier, 1990.

[15] Wikimedia Commons. *Hash function.* 2005. URL: https://commons.wikimedia.org/wiki/File:Hash_function.svg (visited on 08/24/2017).

[16] Wikimedia Commons. *Diagram illustrating how PGP works.* 2012. URL: https://commons.wikimedia.org/wiki/File:PGP_diagram.svg (visited on 08/24/2017).

[17] Philip R Zimmermann. *The official PGP user's guide.* MIT press, 1995.

[18] *RSA 1024-bit private key encryption cracked.* URL: http://www.techworld.com/news/security/rsa-1024-bit-private-key-encryption-cracked-3214360/ (visited on 07/21/2017).

[19] Don Johnson, Alfred Menezes, and Scott Vanstone. "The elliptic curve digital signature algorithm (ECDSA)". In: *International journal of information security* 1.1 (2001), pp. 36–63.

[20] NIST FIPS-PUB. "180-4. Secure Hash Standard (SHS). March 2012". In: *Retrieved [July 2014] from csrc. nist. gov/publications/fips/fips180-4/fips-180-4. pdf* ().

[21] *Bitcoin blockchain illustration.* URL: https://bitcoin.stackexchange.com/questions/35448/is-it-chain-of-headers-rather-than-a-chain-of-blocks (visited on 08/24/2017).

[22] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology — CRYPTO '87: Proceedings.* Ed. by Carl Pomerance. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378. ISBN: 978-3-540-48184-3. DOI: 10.1007/3-540-48184-2_32. URL: http://dx.doi.org/10.1007/3-540-48184-2_32.

[23] Matthew Sparkes. *Tech giant Microsoft accepts Bitcoin payments.* Telegraph. URL: http://www.telegraph.co.uk/technology/news/11286998/Tech-giant-Microsoft-accepts-Bitcoin-payments.html (visited on 04/21/2017).

[24] *Liberty Dollar.* investopedia. URL: http://www.investopedia.com/terms/l/liberty-dollar.asp (visited on 04/21/2017).

[25] *Mixing service*. URL: https://en.bitcoin.it/wiki/Mixing_service (visited on 04/21/2017).

[26] E.J. Fagan. *Bitcoin and international crime [Commentary]*. The Baltimore Sun. URL: http://www.baltimoresun.com/news/opinion/oped/bs-ed-bitcoin-20131125-story.html (visited on 04/21/2017).

[27] Amin Kharraz et al. "Cutting the gordian knot: A look under the hood of ransomware attacks". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2015, pp. 3–24.

[28] Monica J Barratt. "Silk Road: eBay for drugs". In: *Addiction* 107.3 (2012), pp. 683–683.

[29] Lawrence J Trautman. "Virtual currencies; Bitcoin & what now after Liberty Reserve, Silk Road, and Mt. Gox?" In: (2014).

[30] *Script*. URL: https://en.bitcoin.it/wiki/Script (visited on 04/21/2017).

[31] *Multisignature*. URL: https://en.bitcoin.it/wiki/Multisignature (visited on 04/21/2017).

[32] *Blockchain Size*. URL: https://blockchain.info/it/charts/blocks-size (visited on 04/21/2017).

[33] Silvio Micali. "ALGORAND: The Efficient and Democratic Ledger". In: *arXiv preprint arXiv:1607.01341* (2016).

[34] *Namecoin*. URL: https://namecoin.org/ (visited on 04/21/2017).

[35] Vitalik Buterin et al. *Ethereum white paper*. 2013.

[36] Gavin Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum Project Yellow Paper* 151 (2014).

[37] *Solidity stack*. URL: https://ajlopez.wordpress.com/2016/06/05/compiling-and-executing-smart-contracts-1/ (visited on 08/24/2017).

[38] *Solidity Language*. URL: https://solidity.readthedocs.io/en/develop/ (visited on 04/21/2017).

[39] *Remix - Solidity IDE*. URL: https://remix.readthedocs.io/en/latest/ (visited on 04/21/2017).

[40] *Serpent Language*. URL: https://github.com/ethereum/wiki/wiki/Serpent (visited on 04/21/2017).

[41] Donald R. Morrison. "PATRICIA&Mdash;Practical Algorithm To Retrieve Information Coded in Alphanumeric". In: *J. ACM* 15.4 (Oct. 1968), pp. 514–534. ISSN: 0004-5411. DOI: 10.1145/321479.321481. URL: http://doi.acm.org/10.1145/321479.321481.

[42] Yonatan Sompolinsky and Aviv Zohar. "Accelerating bitcoin's transaction processing". In: ().

[43] *Ethash*. URL: https://github.com/ethereum/wiki/wiki/Ethash (visited on 04/21/2017).

[44] Primavera De Filippi and Benjamin Loveluck. "The Invisible Politics of Bitcoin: Governance Crisis of a Decentralized Infrastructure". In: (2016).

[45] Nick Szabo. "Formalizing and securing relationships on public networks". In: *First Monday* 2.9 (1997).

[46] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. "On Bitcoin as a public randomness source." In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 1015.

[47] Vitalin Buterin. *Privacy on the Blockchain*. URL: https://blog.ethereum.org/2016/01/15/privacy-on-the-blockchain/ (visited on 04/21/2017).

[48] Adi Shamir. "How to share a secret". In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[49] Oded Goldreich. *A Short Tutorial of Zero-Knowledge*. 2013.

[50] Elena Mesropyan. *21 Companies Leveraging Blockchain for Identity Management and Authentication*. URL: https://letstalkpayments.com/22-companies-leveraging-blockchain-for-identity-management-and-authentication/ (visited on 04/21/2017).

[51] Lucia Ziyuan. *Web of Trust: ConsenSys Talks Ethereum Future, Presents uPort Blockchain Project*. URL: https://cointelegraph.com/news/web-of-trust-consensys-talks-ethereum-future-presents-uport-blockchain-project (visited on 04/21/2017).

[52] Brianne Rivlin. *Real Estate Meets Ethereum*. URL: https://www.ethnews.com/real-estate-meets-ethereum (visited on 04/21/2017).

[53] Patrick McCorry, Siamak F Shahandashti, and Feng Hao. "A Smart Contract for Boardroom Voting with Maximum Voter Privacy". In: ().

[54] *Proof of Stake FAQ*. URL: https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ (visited on 04/21/2017).

[55] *Etherscan*. URL: https://etherscan.io/chart/gasprice/ (visited on 04/21/2017).

[56] Juan Benet. "IPFS-Content Addressed, Versioned, P2P File System (DRAFT 3)". In: ().

[57] *Swarm*. URL: https://github.com/ethersphere/swarm (visited on 04/21/2017).

[58] David Siegel. *Understanding The DAO Attack*. June 25, 2016. URL: http://www.coindesk.com/understanding-dao-hack-journalists/ (visited on 04/21/2017).

[59] Jeffrey Travers and Stanley Milgram. "The small world problem". In: *Phychology Today* 1 (1967), pp. 61–67.

[60] Henk P. Penning. *analysis of the strong set in the PGP web of trust*. URL: https://pgp.cs.uu.nl/plot/ (visited on 04/21/2017).

[61] Vitalik Buterin. *Ethereum Research Update*. URL: https://blog.ethereum.org/2016/12/04/ethereum-research-update/ (visited on 04/21/2017).

[62] *eWASM Design*. URL: https://github.com/ewasm/design (visited on 07/21/2017).

[63] Jim Manning. *Ropsten To Kovan To Rinkeby: Ethereum's Testnet Troubles*. URL: https://www.ethnews.com/ropsten-to-kovan-to-rinkeby-ethereums-testnet-troubles (visited on 08/24/2017).

[64] *Securify (smart contract verification)*. URL: https://securify.ch/ (visited on 09/01/2017).

[65] *Defending Against Query Selector Injection Attacks*. URL: https://thecodebarbarian.wordpress.com/2014/09/04/defending-against-query-selector-injection-attacks/ (visited on 09/01/2017).

[66] *OpenSSL*. URL: https://www.openssl.org (visited on 09/01/2017).