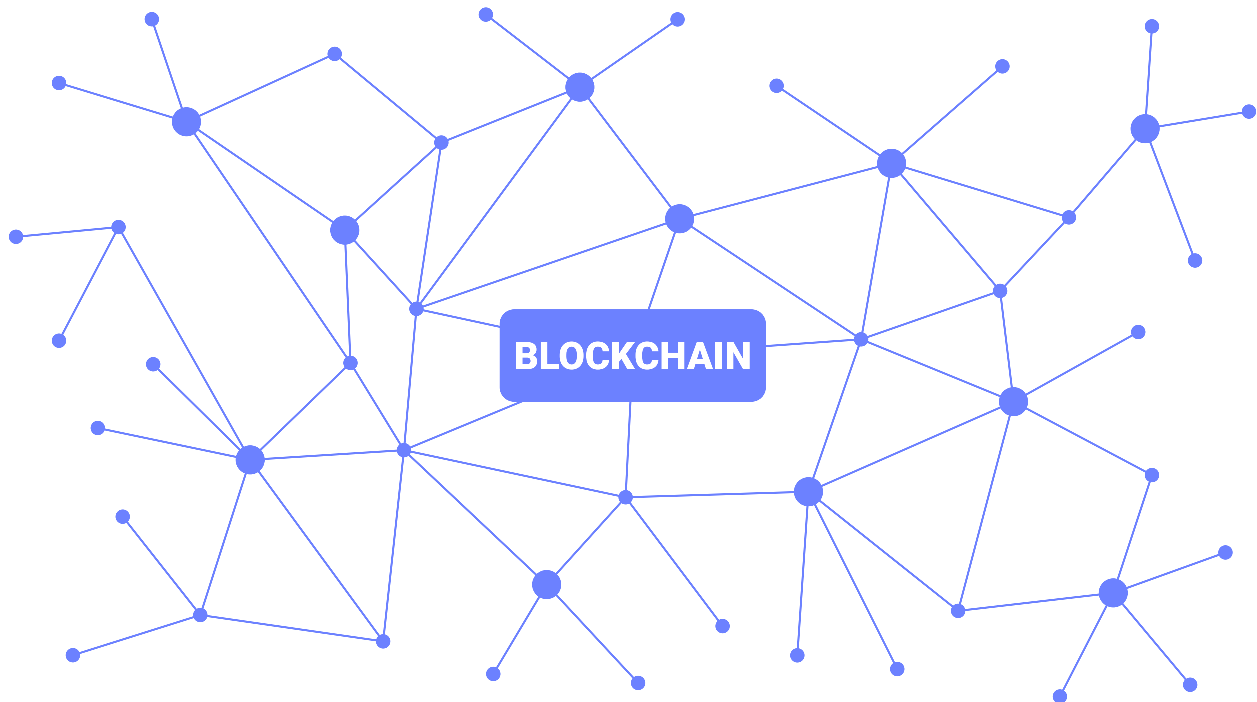
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Authentication using Smart Contracts in a Blockchain

Master's thesis in Computer Systems and Networks

ERIC BORGSTEN

OSKAR JIANG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Authentication using Smart Contracts in a Blockchain

ERIC BORGSTEN
OSKAR JIANG

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Authentication using Smart Contracts in a Blockchain
ERIC BORGSTEN
OSKAR JIANG

© ERIC BORGSTEN, 2018.
© OSKAR JIANG, 2018.

Supervisor: Alejandro Russo, Information Security division, Department of Computer Science and Engineering, Chalmers University of Technology

Advisor: Martin Altenstedt, Omegapoint

Examiner: Carl-Johan Seger, Functional Programming division, Department of Computer Science and Engineering, Chalmers University of Technology

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Authentication using Smart Contracts in a Blockchain
ERIC BORGSTEN
OSKAR JIANG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

A challenge with user authentication using passwords is to ensure that they are kept in a safe storage. The most commonly used approach to achieve this is to have a centralized storage with high security measurements but this puts this storage at high risk of being targeted in an attack. We propose an authentication method using a Smart Contracts in a blockchain as a public key storage to solve this issue. This enables same security standards as traditional authentication, with increased availability and usability. Our results show that this method of authentication is a viable alternative to existing ones.

Keywords: Computer Science, Blockchain, Ethereum, Authentication, Cryptocurrency, Distributed Systems.

／

Contents

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 Aim . . . 2

2 Theory 3
- 2.1 User Authentication . . . 3
  - 2.1.1 Knowledge Based Authentication . . . 3
  - 2.1.2 Ownership Based Authentication . . . 4
- 2.2 Hash algorithms . . . 4
  - 2.2.1 MD5 . . . 5
  - 2.2.2 SHA . . . 5
- 2.3 Cryptography . . . 6
  - 2.3.1 Symmetric-key cryptography . . . 6
  - 2.3.2 Asymmetric-key cryptography . . . 6
  - 2.3.3 Hybrid cryptographical systems . . . 8
  - 2.3.4 Digital signatures . . . 8
- 2.4 Cryptocurrency . . . 9
  - 2.4.1 Merkle trees . . . 9
  - 2.4.2 Blockchain . . . 10
  - 2.4.3 Wallet . . . 10
  - 2.4.4 Distributed Consensus . . . 10
    - 2.4.4.1 Proof-of-work . . . 11
    - 2.4.4.2 Proof-of-stake . . . 11
    - 2.4.4.3 Byzantine Agreement . . . 11
- 2.5 Smart Contracts . . . 12
- 2.6 Ethereum . . . 12
  - 2.6.1 Solidity . . . 12
  - 2.6.2 Gas . . . 13
- 2.7 Web extensions . . . 14
  - 2.7.1 UI pages . . . 14
  - 2.7.2 Content Script . . . 14
  - 2.7.3 Background page . . . 14
- 2.8 Webserver . . . 14

2.9 Public Key Infrastructures 15

3 Related Work **17**
3.1 uPort 17
3.2 Distributed Hash Table 17
3.3 EMCSSL 18

4 System overview **19**
4.1 User 20
4.2 Identity provider 20
4.3 Public Ledger 21

5 Implementation **23**
5.1 Public Ledger 23
5.1.1 Smart Contracts 23
5.1.2 Local Test Node 23
5.1.3 Wallet 24
5.2 User application 24
5.2.1 Registration 24
5.2.2 Login 26
5.3 Web-server 26

6 Verification **29**
6.1 Ethereum 29
6.1.1 Data types 29
6.1.2 Storage Structures 31
6.1.3 Key Types 31
6.2 Authentication 31

7 Results **33**
7.1 Ethereum Storage 33
7.1.1 Data Types 33
7.1.2 Data Structures 34
7.1.3 Key-Types 35
7.2 Full Authentication Process 38

8 Discussion **41**
8.1 Client 41
8.2 Server 42
8.3 Security 42

9 Conclusion **45**
9.1 Future Work 45

Bibliography **47**

List of Figures

2.1 Example of a unbalanced Merkle tree. H denotes a hash algorithm and + denotes concatenation . . . 10

4.1 The proposed protocol for registration and authentication. This process only needs to be done once per identity. . . . 19
4.2 The proposed protocol for authentication . . . 20

5.1 The flow of the registration procedure . . . 25
5.2 The flow of the login procedure . . . 26

7.1 The authentication delay for the final contract that uses the bytes32 type for storage . . . 38
7.2 The authentication delay for the final contract that uses the uint256 type for storage . . . 39
7.3 The authentication delay for the traditional authentication system . . . 39

List of Tables

2.1 The scaling of number of keys in a symmetrical compared to a asymmetrical key system. . . . 7
2.2 The relation between strength, RSA and ECDSA. . . . 8
2.3 The relation between keysizes of different ECDSA curves 8

7.1 Gas cost and read latency associated with the basic data types in solidity 34
7.2 Gas cost and read latency associated with basic data types used to store mismatched data 34
7.3 Gas cost and read latency for different storage structures based on uint256 34
7.4 Gas cost and read latency for different storage structures based on bytes32 35
7.5 Benchmarks for uint256 36
7.6 Benchmarks for bytes32 37
7.7 Authentication latency measured in milliseconds 38

1 Introduction

An end user in the current digital society consume content and services from multiple websites daily. These websites often want to identify the user to for example be able to provide premium content to paying customers, tie a physical person to an online identity or serve more personalized content. This means that the user in some manner needs to authenticate themselves towards the service.

The most common authentication method is using a user identifier, such as an username or email address, that will act as the user's identity on that service and a password which is used to assert that the user is who he claims to be. This is a good solution because of its simplicity. Although, to prevent a single compromised account or lost password jeopardizing the rest of a user's online identities it is recommended to use different passwords for each and every service the user is registered on. This is a small problem if the user is registered to only a handful of services but can quickly become a problem when someone needs to remember details for dozens of services.

Services such as *Lastpass* [35] and *1Password* [34] are password managers providing browser extensions able to generate strong and unique passwords for every service. Furthermore, it stores them in an encrypted database and automatically enters the authentication details into appropriate fields at a given service. This means that the user only needs to remember one password, the one that the password database is encrypted with, while still reaping the benefits of strong unique passwords for each service. A big drawback of this approach is that if the password that the database is encrypted with gets compromised then all accounts stored in the database are compromised as well.

Another approach of solving the problem of having to remember several passwords is Single Sign-On (SSO). SSO systems take a different approach than the previous mentioned services, instead of building on top of the old username and password system they provide the user with the ability to sign in to a service by authenticating themselves towards a third party called an identity provider. In practice this means that any service can add a button to their authentication page allowing their users to login using for example their Google account. The button will take the user to the login page of Google where the user can login using their Google details. When authenticated the user will be returned to the original service where the user is

signed in as the identity linked with that Google account. The SSO approach has many benefits such as user-friendliness, simplicity and the ability for smaller services to implement standardized safe authentication solutions based on big actors such as Google. A major drawback with the SSO-solutions that exists today though is that the end-user needs to trust the identity provider to protect their privacy. Another drawback is the availability of the system, if the identity provider goes down there will be no way to access the accounts on various services where that identity provider has been used as authentication.

1.1 Aim

We aim to design a proof-of-concept system utilizing a blockchain storage for user authentication, solving the troubles regarding weak passwords while ensuring privacy. The usability of this approach will be further analyzed, especially with regard to latency from both a user's perspective and a server owner's. What are the strengths and weaknesses of such a system compared to a traditional one?

2

Theory

In this section terms and concepts, that are important for the understanding of our system, are described.

2.1 User Authentication

Authentication, not to be confused with authorization, is the process of verifying one's identity. This can be utilized in a system where we want to associate an identity with some resources and make sure that no other identities have access to those resources. To achieve this, a method of verifying the identity of the user is needed.

2.1.1 Knowledge Based Authentication

One way of verifying one's identity towards another party is some knowledge that is kept secret between oneself and the authenticating party. A password is a common way of implementing this method in practice. If a password can be communicated safely between those two parties without any interruption or eavesdropping, one's identity can be verified using authentication by knowledge.

The big challenge with passwords is to find one that is both secure and easy to remember. Given an infinite amount of time, any password could theoretically be revealed simply by guessing combinations of characters since a password has to consist of a finite amount of characters. Given that people tend to choose passwords that are easy to remember, the guesses could even be narrowed to character combinations that make sense to a human being. There is also the possibility that the password could be intercepted and stolen by a malicious entity when the user enter it into the computer, when it traverses the web or when it arrives at its destination. A compromised password in any of those stages could result in a compromised identity.

2.1.2 Ownership Based Authentication

Another way of verifying one's identity is to associate the true owner of the identity with a unique item. Examples of such items are smart-cards and Radio-Frequency identifier (RFID) tags. This method of authentication requires devices that can read them reliably. Secure storage also have to be ensured since the possessor of the item is always assumed to be the true owner of the identity.

A reliable read of an object can be difficult in practice. Constructing physically identical items could be done but is not required to circumvent the security, an object that is perceived as identical when reading it would probably be sufficient. Therefore, items that are nearly impossible to replicate but can be read with a very high accuracy is ideal for this method of authentication. The devices also tend to be a more expensive alternative than a password system.

There is a similar method of authentication that is called *inherence based authentication*, where the identity is asserted by something that the user is, e.g. bio-metrics such as fingerprints or eyes.

2.2 Hash algorithms

A hashing algorithm aims to be a one-way operation that takes an input of arbitrary length and provides a fixed size digest of the data input. Reversing this operation should in practice be unfeasible. This digest is basically a deterministic mathematical summary of the data input which means that identical input data will generate the same output. Good hashing algorithms should have low hash collision rate, meaning it should be hard to find two different data inputs that generate the same output digest. Furthermore, it should also be resistant to what is known as a pre-image attack. This is an attack where the attacker, given a message digest, is able to calculate a different input that will result in the same digest. Finally, a good hashing algorithms should be resistant to second-pre-image attacks. This attack is similar to the pre-image attack but the input value is known instead of the output digest.

The properties of hashing algorithms make hash digests of password preferable when storing passwords in databases, because the passwords will be secure even if an attacker compromises the database.

Another use of hash digests is to assert the integrity of some data, i.e. that the data has not been modified. This is accomplished by publishing the data together with a hash digest representing that data. The user can then verify the integrity of data by calculating their own hash digest and comparing it to the one published by the creator.

2.2.1 MD5

The Message Digest 5 algorithm or MD5 was created by Ronald Rivest in 1992 in an effort to improve his 1990 algorithm MD4 [45]. MD5 has several improvements in terms of security compared to MD4 which was primarily created for speed [44]. In 2005 however, X. Wang et al. [50], came up with a collision attack which basically left both MD5 and MD4 useless in terms of data fingerprinting. Their attack was able to, in less than an hour of computation time, find a MD5 digest collision with different inputs. The attack was also able to find a MD4 collision in less than a second. Both MD4 and MD5 produce an 128-bit long digest.

2.2.2 SHA

Secure Hashing Algorithm or SHA is a family of hashing algorithms. The first published version, SHA-1, was released in 1993 and was based on the work by Ronald Rivest, described in Section 2.2.1 [24]. It was developed to become a standard hashing algorithm used by American government departments. SHA-1 takes input shorter than 2^{64} bits and produces an 160-bit digest. SHA-1 was however broken in 2005 by X. Wang et al. [49], they found a method of finding hash collisions in at a most 2^{69} hash operations. This was a large improvement compared to the theoretical maximum, which would require 2^{80} hash operations. Stevens et al. [46], performed the first full attack on SHA-1, they performed $2^{57.5}$ hash operations on their 64-GPU cluster and it took them 10 days to find a collision. The corresponding GPU time would cost approximately $2000 to rent and therefore they recommended SHA-1 to become deprecated [46].

SHA-1 was super-seeded by SHA-2 which basically is a group of hashing algorithms. The two main ones are SHA-256 and SHA-512 with the main difference being their input limit, 2^{64} and 2^{128} bits respectively, and the length of their produced digest which are 256 and 512 bits [25]. SHA-2 is at the moment considered as a sufficiently secure hashing algorithm by the European Union Agency for Network and Information Security [29].

The latest development in the SHA family is the SHA-3 or the keccak algorithm. The keccak algorithm was chosen as the next algorithm in a competition hosted by NIST which it won. Keccak is not based on the previous algorithms in the family but instead on something called sponge construction. This allows keccak to provide the output digest to be of arbitrary length, the 4 varieties of the keccak algorithm that is included in SHA-3 is keccak448, keccak512, keccak768 and keccak1024 which have fixed digest lengths of 224, 256, 384 and 512 bits respectively [30].

2.3 Cryptography

The goal of cryptography is to provide data confidentiality, i.e. protect data from being read by other parties than the intended receiver. This is achieved by transforming the data in a way so it becomes useless for anyone trying to intercept it and impossible for anyone to change it without the receiver noticing [27]. Secure communication can be established on the Internet using this. In addition, parties can be sure that they are in fact communicating with the intended receiver and that nobody else can extract the content of their communication. There are two main types of cryptographic systems and they will be described in the following subsections.

2.3.1 Symmetric-key cryptography

The main principle behind symmetrical-key cryptography is that the sender and receiver of the encrypted message share a common secret that is both used for encryption and for decryption. This is both an advantage and a disadvantage because while it keeps the complexity of the cryptographic algorithm low, which in turn will make it fast and efficient, it also requires that this common secret is either pre-shared or shared to the receiver securely. This introduces a catch-22 situation, cryptography is used to protect the data transmitted from third parties but to be able to encrypt your data you need to transmit the key securely without a third party being able to intercept it. You could rely on pre-shared keys that you generate and share in person but this makes it hard to establish encrypted communications with new parties that you have never met.

Another drawback with symmetrical encryption is the amount of keys that users are required to store. If we take a group of n members for example, everyone wants to be able to conduct private communications with everyone else in the group separately. This would require that each member in the group to have a shared secret with every other member in the group, this would result in a total of n^2 keys in the group, which scales quite bad as the group grows.

2.3.2 Asymmetric-key cryptography

To solve the problems related to symmetric encryption, asymmetric-key cryptography was invented. Instead of each communication pair sharing a common secret, each user of asymmetric-key cryptography generates a key pair. This pair consist of a private key that the user always must keep secret and a public key which they can share freely in completely insecure channels. When someone wants to send an encrypted message they encrypt their message using the *receiver's* public key, the receiver will then use their secret key to decrypt the message and can reply using

Number of members	Symmetrical keys	Asymmetrical keys
1	1	2
2	4	4
3	9	6
4	16	8
5	25	10
10	100	20

Table 2.1: The scaling of number of keys in a symmetrical compared to a asymmetrical key system.

the public key of the original sender.

This solves the problem regarding key sharing that exist in symmetric-key cryptography because if someone wants to send encrypted messages to another person they can just send their public key in an email to the receiver and request that they do the same, when both possess the others public key they can encrypt all their messages.

For small groups or point-to-point communication an asymmetrical-key approach would require more keys to be stored, a total of $2n$, but the number of keys will scale linearly while a symmetrical approach will scale exponentially. This can be seen in Table 2.1.

One major drawback with asymmetrical cryptography compared to symmetrical cryptography is the additional mathematical complexity which causes larger overheads and therefore lower performance [43].

When comparing the relative strength of different cryptographical algorithms a metric called *security strength* is used. This metric is a calculated value of how large the key size would be in a symmetrical cryptographical algorithm to achieve the equivalent cryptographical strength. This makes it possible to compare for example RSA and Elliptic Curve Digital Signature Algorithm (ECDSA) which are based on different math problems which vary in how computationally hard that problem is to solve. An example of such a comparison can be seen in Table 2.2 [23, 28]. This table shows that both RSA and ECDSA need longer keys than a symmetrical algorithm to achieve the same level of strength. It can also be observed that RSA needs much larger keys than ECDSA to be equivalent in terms of cryptographical strength. ECDSA can use a compressed form as well as an uncompressed form to represent the public keys. The normal uncompressed form is double the length of the private key while the compressed form is the length of the private key plus 8 bits. The key sizes for some example ECDSA curves is described in 2.3.

Strength	RSA (modulo size)	ECDSA curve name
112 bits	2048 bits	secp224r1
128 bits	3072 bits	secp256k1
192 bits	7680 bits	secp409k1
256 bits	15360 bits	secp521r1

Table 2.2: The relation between strength, RSA and ECDSA.

ECDSA Curve	Private key	Public Key uncompressed/compressed
secp224r1	224 bits	448/232 bits
secp256k1	256 bits	512/264 bits
secp409k1	409 bits	818/417 bits
secp521r1	521 bits	1024/529 bits

Table 2.3: The relation between keysizes of different ECDSA curves

2.3.3 Hybrid cryptographical systems

In real life systems a combination of symmetrical and asymmetrical cryptography systems is often used, this is due to how their advantages and disadvantages complement each other. Symmetrical algorithms are used to provide raw data encryption speed while asymmetrical algorithms are used to securely transfer the shared secret to the receiver. This also has the additional benefit that the shared secret can be generated on a session basis, this means that if that key is somehow compromised then it is only the current session that is compromised, not previous and future ones.

An example of a hybrid cryptographical system is TLS which is the protocol used to secure web traffic on HTTPS-enabled sites [26].

2.3.4 Digital signatures

Digital signatures fulfill the same purpose as their analog counterparts. It relies on a combination of asymmetric cryptography and hashing algorithms to provide a way for message recipients to assert the authenticity and integrity of the message.

A signed message is created and sent by doing as follows:

1. Calculate a hash digest of the message with an hash algorithm known to the receiver.

2. Encrypts the hash digest, but instead of using the receiver's public-key use the sender's private-key.

3. Send this encrypted hash digest along with the unencrypted message.

When the receiver then wants to verify the received message they will follow the

following steps:

1. Decrypt the received signature with the public-key of the supposed sender.

2. Calculate a hash digest of the received message.

If the decrypted signature and the hash is equal then the receiver has verified both the authenticity and the integrity of the message. If on the other hand the decrypted signature is not equal to the calculated hash then either the sender is not who they claim to be or the message has been changed or corrupted during transmission.

Note that unlike encryption using asymmetric-key cryptography, where you only can encrypt your message towards one public-key at a time, the sender can sign their message once and everyone that have the public-key associated with the sender can verify the authenticity and integrity of the message.

2.4 Cryptocurrency

In contrast to fiat currency, cryptocurrencies do not utilize a central authority for validating transactions involving the currency [41]. Instead, it relies on nodes in a network to validate the transaction using computing power. This process is also commonly known as mining. The transaction log is then stored in a public ledger which contains all transactions that have occurred in the system since its genesis. The nodes that validate the transaction are awarded with cryptocurrency.

2.4.1 Merkle trees

Merkle trees is a data structure based on binary trees. In contrast to a binary tree, only the leaves are used to store data and the leaf nodes are the only child to their parents. The leaves' parent nodes contain a hash of their leaf nodes. The next generation of nodes however each have two child nodes and contain a hash of the hash that their child nodes contain. If the tree is unbalanced because there is an uneven amount of leaves, the lonely leaf's hash will be used twice. The advantage of this data structure is that the integrity of the whole tree can still be verified even though redundant leaves has been discarded. An example of a Merkle tree can be seen in Figure 2.1.

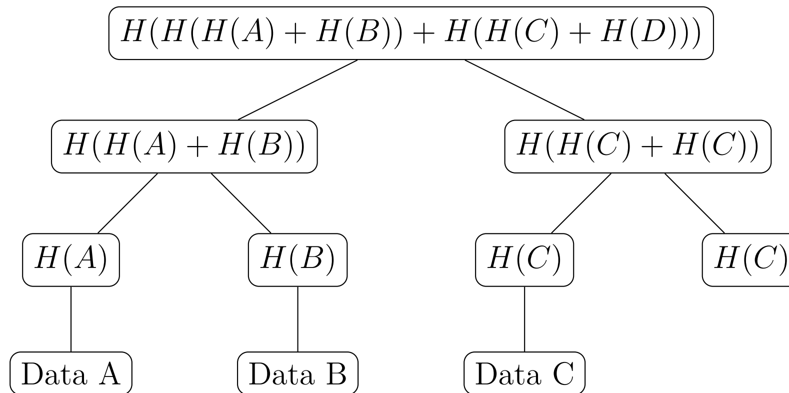


Figure 2.1: Example of a unbalanced Merkle tree. H denotes a hash algorithm and $+$ denotes concatenation

2.4.2 Blockchain

Cryptocurrencies rely heavily upon cryptography to ensure correctness in the transaction log. They utilize a blockchain to organize their ledger. A blockchain is as the name suggest a chain of blocks, where each block is numbered and contains a hash digest of the previous block. This means that if one were to start at the original block in the chain, the origin block, then every block could be verified by the next block in the chain. It is also immutable which is crucial for their role in cryptocurrencies.

2.4.3 Wallet

A cryptocurrency wallet essentially consists of a cryptographic key pair. The wallet address, that is used as an identifier when other people want to send currency to you, is generated from the public key and every transaction you want to make is signed with your private key [38]. This means that counterfeiting transactions are unfeasible and everyone can verify the legitimacy of the transaction easily using the sender's public key. The account balance is calculated from all the transaction in the ledger that you were involved in.

2.4.4 Distributed Consensus

Storage using a distributed ledger implies that every entity with a copy of it may potentially have a different fork, i.e. state, of the blockchain. Therefore, it is important that every node in the network agree, at some point in time, about which the correct version is.

2.4.4.1 Proof-of-work

The consensus algorithm used in for example Bitcoin is called proof-of-work [42]. The general idea is to prove the validity of your ledger by making calculations to achieve a result that fulfills certain requirements. In the case of Bitcoin the computation aims to find a nonce that when hashed, returns a hash value lower than a globally defined target value. Due to the one-way nature of hashing algorithms, where it is unfeasible to calculate a value that would result in an acceptable hash, the optimal method of finding such values is by simply iterating through numbers and hashing them. This means that the only way of increasing the chances of finding a nonce with the correct properties is to increase the computer's hashing power. Thereby the term proof-of-work, a computer possessing such a value must have done work to obtain it. Since every node in the network that are mining are working on their own proof-of-work, they need to agree about whose to chose. Therefore, nodes that have found an acceptable hash, broadcast this and the other nodes in the network can easily verify that this is correct by making the calculation themselves. If a longer valid blockchain is propagated to nodes, they always chose this one instead of their current one. The longest blockchain, which also is the one with the most amount of computation work put into it, will eventually be agreed upon as the main blockchain by every node in the network. Alternative versions of the main one, e.g shorter blockchains that exist at nodes that the longest one have not been propagated to, are called forks.

2.4.4.2 Proof-of-stake

In 2012 S.King et al. suggested another method of reaching consensus called Proof-of-stake [36]. The term coin age A_c is crucial for this concept. It is defined as the number of coins N_{coin} times the number of time units a user has had it in its possession T_{coin}. To generate a new block a user needs to do a so called coinstake transaction. The sender and receiver of this transaction is the same, namely the user himself. Instead of working towards a target value using computation power, the target is instead reached using coin age. The more coin age, the faster the block can be expected to be generated. In addition, the blockchain with the highest amount of invested coin age is considered as the valid one. Thereby, we end up with a system completely regulated using stakes and the authors argue that this could replace proof-of-work.

2.4.4.3 Byzantine Agreement

The Byzantine Generals problem was a distributed consensus problem described by Leslie Lamport et al. in the year 1982 [37]. Arbitrary systems in a network are, in this case, described as generals in an army. The problem is to reach an agreement with 100% confidence in despite the presence of dishonest generals that will do

everything they can to stop an agreement for being made. They showed that this is possible given $3m + 1$ generals, where only m generals are dishonest. The protocol assumes that all the generals receive messages from all other generals about their intentions. If there is one intention that constitutes a majority of all the messages, the generals will agree upon that. And thus, agreement has been achieved.

2.5 Smart Contracts

Contracts state an agreement between parties stating the obligations towards one another. If any of the parties breaks this agreement, it usually renders in some sort of unwanted consequences for the party breaking it. Physical signatures has traditionally been used to link identities to specific contracts. The idea of Smart Contracts was first introduced by Nick Szabo in 1997 before the existence of any cryptocurrencies [47]. He proposed an idea, using recent advancement in computer technology, of integrating contracts into software and hardware to be able to come to agreements digitally. He stated three objectives for Smart Contracts for a successful application of it; observability, verifiability and privity. Observability and verifiability address the issue of both being able to see and verify the other parties' performance of the contract properly which could easily be achieved in the context of traditional contracts. Privity is also necessary to prevent third parties to control the performance of the contract or the contract itself, rendering the very meaning of it useless.

2.6 Ethereum

Ethereum is an open-source protocol of a distributed network based on blockchain technology [1]. Ethereum support both Smart Contracts and simple monetary transactions of the protocol's cryptocurrency Ether. There are *externally owned accounts*, that are controlled by private keys, and *contract accounts*, that are controlled by the code in the previously mentioned Smart Contracts, in Ethereum.

2.6.1 Solidity

Solidity is a statically typed, Turing-complete, contract-oriented high-level language [2] created and developed by the Ethereum project. The Solidity code runs in an isolated environment known as the Ethereum Virtual Machine (EVM) [13]. The executed code does not have access to any other process on the machine and only has limited access to other Smart Contracts. Contracts written in Solidity have some similarities with objects in object-oriented languages such as Java [19]. The contracts support state variables as well as functions and events etc. To deploy

the contracts written in solidity to the blockchain they first needs to be compiled to bytecode, this is done with the compiler *solc*, which also supports additional operations such as code optimization, output EVM-assembly, gas cost estimation etc.

In solidity the following basic data types are available

- **Boolean** — Basic Boolean, can be either true or false.
- **Integer** — Integers have a fixed-size between 8 and 256 bits, the size must be divisible by 8. There are both signed and unsigned Integers.
- **Address** — Address is a 20 byte type that is designed to store an Ethereum address.
- **Byte** — Byte is a fixed-size byte array that can be declared to have a size between 1 and 32 bytes.
- **Bytes** — Bytes is a dynamically sized byte-array, should only be used for raw data of arbitrary length due to its high cost.
- **String** — String is a dynamically sized byte-array, should only be used for UTF-8 encoded data of arbitrary length due to its high cost.
- **Enum** — Basic Enum type that can adopt predefined values.

These types can then be used in conjunction with C-like Structs to create advanced data structures. There is also the type *mapping* which allows you to bind a value to another, basically creating a dictionary.

2.6.2 Gas

Every externally owned account has a balance that is measured in Ether. Every transaction, including contract functions or deployments, costs a certain amount of gas to execute. Gas is a measurement of how much computation is required to perform the actions specified by the transaction. The users that sends the transaction can specify how much Ether they are prepared to pay per gas but if no price is specified then the current average gas price will be used. This decouples the transaction cost from the volatile cryptocurrency and also creates an auction-like system where users prepared to pay more will have their transactions handled more quickly [7].

2.7 Web extensions

Web extensions are plugins that are able to modify or add functionality to modern web browsers [32]. They consist of HTML, CSS and Javascript files and has similar behavior to a web application, but runs as an embedded application in the browser. The extension uses the manifest to declare global information such as the name of the extension and what permissions it has. Web extensions offers a local storage that can be used to store information.

2.7.1 UI pages

The extension has its own UI that can be activated by the user. It consists of a view and script files to control the behavior of said view. These scripts are only executed when the UI is active.

2.7.2 Content Script

The content script is executed in parallel with the current website and is as the name suggests a script. It is the only part of the extension that can interact directly with the website and can pass messages to the extension if needed.

2.7.3 Background page

The background page can either be a persistent background page or an event page. The former is always running in the background and the latter only execute at certain events. It consists, just as the UI pages, of a view and a script but can function even when the extension is inactive. As it runs as part of the extension it can work as a link between the extension and the content script.

2.8 Webserver

A webserver provides resources to clients using the HTTP protocol [40]. Resources can be targeted using a URL and is provided given that the client is authorized and that the resource exists. The resources are usually stored in a database and often combined with a back-end application to handle the logic involving requests and database interaction. Webserver can refer to either the physical machine holding these resources or the software handling the client requests, or both. In this context

the web server is the software, as the machine it runs on is assumed to be of less importance, but more importantly the machine interchangeable.

2.9 Public Key Infrastructures

A public key infrastructure (PKI) is commonly used to assert integrity and ownership of public keys [17]. It is necessary when two parties wishes to communicate either through encrypted messages or signed messages but wants to verify the identity of the receiver. Assuming that the two parties have access to each other's public keys, they can still not be sure about the ownership of that key e.g A can not know if B's public key really is B's. To solve this a Certificate Authority (CA) is used. A CA is a third party that specializes in validating the connection between an identity and a public key by issuing a signed certificate. This certificate verifies the connection between an identity and a public key. Such certificates are called SSL certificates and can be bought through trusted CAs [22]. A certificate can also be self-signed, meaning that the one holding the private part of the keypair validates its own public key.

3

Related Work

Although the field of cryptocurrency and their practical applications are a relatively new, at the time of writing, there does exist some attempts at utilizing it for authentication. In these section we describe concepts that also attempts to solve availability and/or privacy issues that are involved with current authentication solutions.

3.1 uPort

uPort is a platform that utilizes Ethereum to create self-sovereign identities [39]. The main idea involves keeping your identity stored on your mobile device in form of a private key. The private key could then be used for interactions involving your identity, in this case specifically involving Smart Contracts or transactions in the Ethereum blockchain. Unless the mobile gets stolen, the true owner of it also has total control over the identity linked to it. uPort addresses problems such as this with recover mechanism built into special contracts called the controller contract and the proxy contract. While the designers have suggested user authentication as a possible use cases, uPort remains as a platform for decentralized identities that leaves specific applications of it up to the user.

3.2 Distributed Hash Table

In 2008 A.Takeda et al. proposed a solution to peer-to-peer authentication using the web of trust. They suggested the public keys being kept in a distributed hash table (DHT), to minimize the storage on each node [48]. The keys in it could then be used to verify messages between one another. The storage could, just as in a blockchain, be stored in a decentralized manner with the main difference being that no node has access to all data in the storage in a DHT. While this does require less storage on the nodes, this does create challenges that are not associated with the blockchain technology such as leaving nodes and consensus.

3.3 EMCSSL

Emercoin is a cryptocurrency, built on the code of Bitcoin, that specializes in name-value storage [5]. One technology that relies on this is the EMCSSL that utilizes the storage to store hash sums of SSL certificates [33]. These certificates can be used for authentication at an arbitrary web service and will be highly available since it is stored in a blockchain. The aim for this concept is to prevent man-in-the-middle attacks that are possible during current implementations of authentication using SSL where certificates are issued by CAs. If an attacker manages to intercept the traffic he can switch the certificate to their own. EMCSSL prevents this by having the user sending the certificate to the web service for authentication, the service then in turn checks that this certificate in fact matches the one in the Emercoin blockchain and that it has not been substituted.

4

System overview

To solve the issues regarding privacy, remove the need for remembering passwords and prevent weak passwords from being used for authentication, we propose a concept called Distributed Single Sign-On (DSSO). In this system there are three entities: The user, the identity provider and the public ledger. A general overview of the flow is illustrated in Figure 4.1 and Figure 4.2. The user and the ledger communicate only when changes related to an identity is requested e.g they do not need to communicate at read-operations. Much like in a PKI, the stored identity is the public part of an asymmetric keypair to ensure that anyone with access to the ledger can verify the authenticity of a signed message sent by the owner of the keypair. This is the cornerstone of the authentication process since it also enables anyone to verify the integrity of a message from the user, which then can be used to establish an authorized session.

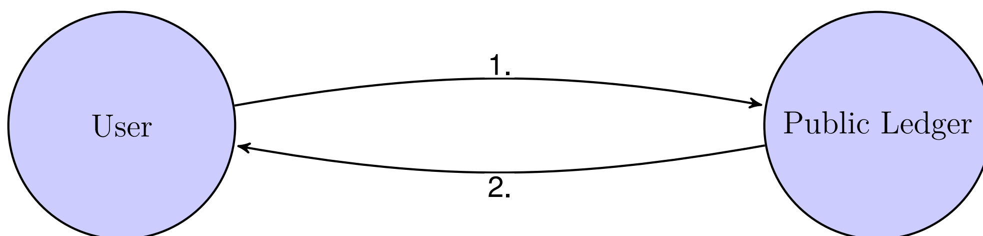


Figure 4.1: The proposed protocol for registration and authentication. This process only needs to be done once per identity.

1. User writes their public key to the ledger
2. If the write is successful, a pointer to the written data is returned.

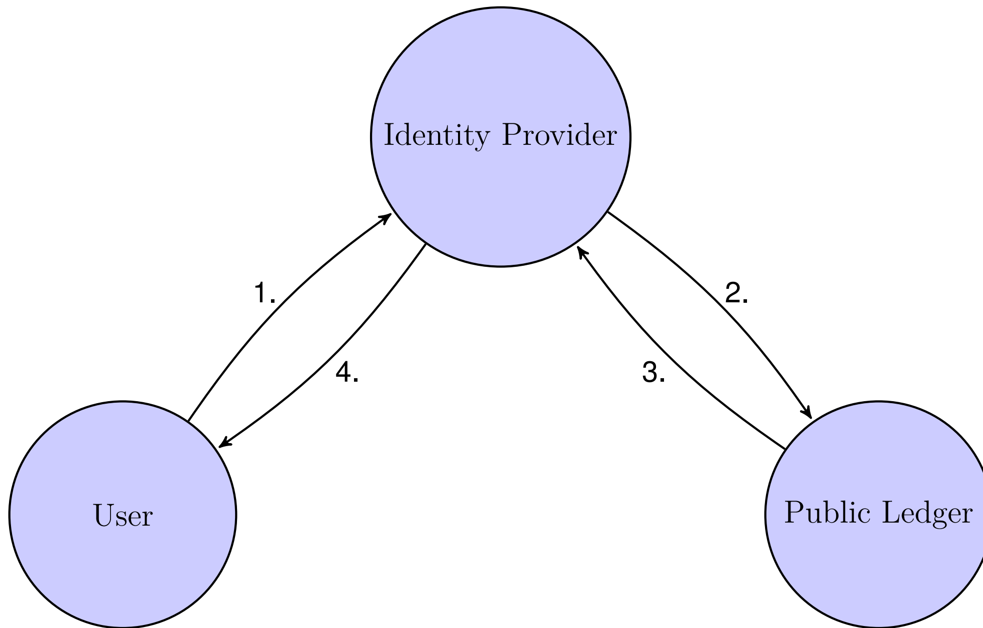


Figure 4.2: The proposed protocol for authentication

1. User attempts to authenticate using a signed message and a pointer to the public key that can be used to verify the authenticity of the message
2. Identity provider checks in Public Ledger for public key
3. Identity provider verifies the signature the user sent
4. Identity provider responds to user if authentication was successful or not.

4.1 User

The user is the entity that wants to authenticate itself towards a service. To achieve this, one needs to store it's public key so that the identity provider can access it. As previously mentioned, we propose storing it in a public ledger to ensure maximal availability and flexibility. A reference to this transaction, together with a signed message, can then be used to verify one's identity.

4.2 Identity provider

This is the entity that can authenticate the user. In most cases the identity provider will work together with a resource owner or a service provider where the user will be redirected to after the authentication.

4.3 Public Ledger

The public ledger uses a blockchain to store information in a distributed manner while maintaining consistency. The validity of the data stored in a specific block increases with the number of blocks in the chain after it, provided that the node is synchronized with the rest of the network. Information from the ledger can be provided locally, by being part of the network, or remotely, by requesting it over the Internet from a node that is part of the network.

5

Implementation

This section describes our implementation of the system that was suggested in the previous chapter. Specific technologies that was needed for the implementation are explained as well the reasoning for choosing these particular ones.

5.1 Public Ledger

Our implementation is based on Ethereum which is the biggest cryptocurrency, measured by market capitalization, that supports Smart Contracts [3, 10]. The contract storage is also part of the ledger and could be used to store information to the same extent as transactions. They also add the possibility of using distributed applications (DApp). The main motivation for choosing Ethereum is the amount of frameworks and support that exists due to the popularity of it. In addition, it supports customized functionality to a higher extent than for example Bitcoin.

5.1.1 Smart Contracts

The functionality to deploy Smart Contracts to the blockchain is included in the EVM and our implementation of a Smart Contract is written in Solidity. The contract includes a function to extract the public key from it, available for anyone and functions to change the public key and destroy the contract, which are only available for the owner. The role of the contracts in our implementation is similar to the ones of self-signed certificates.

5.1.2 Local Test Node

Ganache is an application used for running an Ethereum test network, and offers a graphical interface for exploring the blockchain [11]. This was used during development since the main Ethereum network is both slow and clumsy to work with when testing. Ganache provides an easily accessible test environment including accounts

with a predetermined amount of Ether and lowered mining difficulty, enabling faster transactions. It also provides functionality to easily reset the network which further increases the functionality during development.

5.1.3 Wallet

Metamask is a web extension that acts as a wallet towards an arbitrary Ethereum network [15]. Therefore, we have chosen to use it to interact with the Ganache application to be able to deploy contracts on the local network and receive the contract identifier when the transaction has been performed. But it also allows the user to interact with the main network, either through a local main network node or through their servers. This is considered secure even if the option of using their servers is used because the key storage is encrypted and stored locally and all the transactions are signed before they are sent to their servers.

5.2 User application

Since we needed a lightweight application to catch events from the active website, perform computations based on events and store data for the user, a Web Extension was chosen for this purpose. A web extension offers the ability to create a user-friendly interface to automate the process of authentication as much as possible. To perform all the cryptographical tasks that we need, such as generating keys and creating signatures, we use the jsrsasign library [14].

5.2.1 Registration

The user utilizes the web extension that interacts with Metamask to deploy the contract into the Ethereum blockchain. The extension provides the contract that to be deployed to the network and Metamask ensures that the transaction gets signed and that the required funds exist. It then properly propagates the transaction to the Ethereum network. The data flow is described further in Figure 5.1. In our case we interacted with a test network hosted by the Ganache application instead of the main Ethereum network. When the contract is part of the transaction log, a user can be considered registered as he now has access to a contract identifier of the contract containing his public key. Anyone with access to the ledger can thus fetch it given the contract identifier.

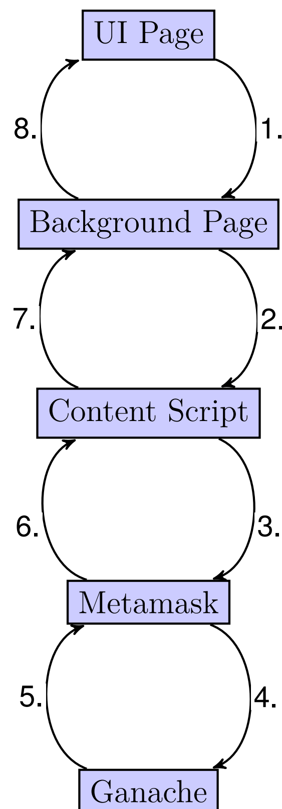


Figure 5.1: The flow of the registration procedure

1. A 'Deploy' command is sent by the user
2. Script for contract deployment is passed to browser
3. The script, that is the contract that includes the public key, is injected to the current website for Metamask to detect
4. Contract is deployed by Metamask after payment and confirmation
5. Contract identifier is returned when deployed
6. Identifier appears at website
7. Identifier is detected and pass back into extension
8. Contract Identifiers saved in the extension and can now be used as a pointer to the public key

5.2.2 Login

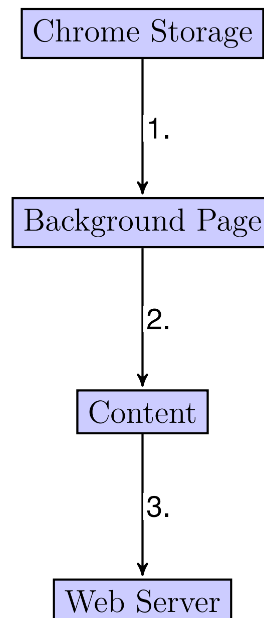


Figure 5.2: The flow of the login procedure

1. Contract identifier and private key is retrieved from the extension storage
2. A JSON-object with the Contract identifier, the time-stamp and the signature of the time-stamp is forwarded to the content script
3. The payload is included in a POST-request towards the web-server we want to authenticate towards

To be able to sign in using the DSSO, the user utilizes the data stored in the extension to compute the data needed for authentication. The extension starts to compute this when it detects that the user has interacted with the login button provided by the resource owner. The JSON-object needed for authentication is generated by the extension and passed to the resource owner using a POST request. An example of such an object is displayed in Listing 5.1. The time-stamp is validated by the server. In our implementation we have chosen a window of 30 seconds when it is considered valid. This leaves a huge room for latency that is caused by transmission and by the clock skew, while leaving a relatively small window for replay attacks.

5.3 Web-server

The web-server we implemented for testing our solution was built using Django. It is a full-stack framework written in Python that controls everything down to database interaction up to the view [31]. Writing the web-server in a different language than

```
1 {
2     "contractID": "0x568abea6429f7f19b5b10b7b4de32e8f9d18d972",
3     "timestamp": "1524078351",
4     "signature": "3045022100afb89050fc2b4d05d5b3541c0f64c7a
5                   fdba1381a7654e12fcabb7f97bc4349b002205864
6                   4a1bd917e2459691c28f13f772964d9cf3d1e2673
7                   2c359d6ea3237a67627"
8
9 }
```

Listing 5.1: Example of the data that is passed in the POST-request used for authentication

the user application was preferred to ensure that the functionality was not language specific.

The web-server was created to demonstrate how an arbitrary website can provide this method of authentication by fetching the public key from the ledger and exposing this option in the view. It handles user authentication by verifying the signature, and the existence of a local identity bound to the global identity. After that, the user can be considered authenticated and a session can be established granting access to the resource as that user. For signature verification the Cryptography library for Python by Python Cryptographic Authority was used [21].

6

Verification

In this chapter the verification process of our system is described. We based our process on the system that we built. The goal was to benchmark crucial parts of the system in order to be able to evaluate the performance as well as the usability for the whole system.

6.1 Ethereum

Tests have been carried out to find the optimal approach to store the user's public key in an Ethereum Smart Contract written in Solidity. Data type, data structure and key type all affect the price of storing a public key as well as the latency for fetching it from the contract. An example of a benchmark that was used to estimate the gas required to deploy a contract can be found in Listing 6.1. The benchmarks were written in Javascript. Read latency was measured using *process* in node.js [16] taking the minimum time of 100,000 attempts. All the performance tests has been carried out on a system with the following specifications:

- CPU: Intel® Core™ i5-3570K @ 3.80 GHz
- Memory: Corsair 8GB (2x4GB) DDR3 CL9 1600MHz
- Storage: Intel® SSD 330 Series 120GB, SATA 6Gb/s

6.1.1 Data types

The basic types in Solidity have different properties which make them suitable for different tasks. Types such as *uint* support data of different sizes such as *uint256* and *uint8*. We have evaluated if it is more advantageous to store data in an undersized type (split the data over several smaller data types) or oversized type (using largest possible data type to store data that does not necessarily need that much space).

```
async function estimateGas(contract, args) {
    const abstractContract = web3.eth.contract(
        contract.abi
    );
    const deploymentData = abstractContract.new.getData(
        args, {data: contract.bytecode}
    );

    const gas = await web3.eth.estimateGas({
        data: deploymentData
    });
    return gas;
}
```

Listing 6.1: Javascript function that estimates gas cost to deploy a contract

Storing data in the blockchain permanently can be quite costly in terms of gas, we have therefore tested which data type that is most cost-efficient in terms of storage.

The tests of the basic data types were carried out by writing Smart Contracts that contain the declaration of the data type in question, these contracts where then analyzed by the solc compiler in terms of gas cost. Furthermore, the contract was also analyzed by the web3 *estimateGas* which includes the cost of the complete transaction. The estimation was then confirmed by actually conducting the deployment transaction.

When the tests above were complete we proceeded by testing the gas cost for assigning data and modifying said data in the same manner. The values assigned to the data types to test them was chosen to be as similar to each other as possible in terms of size to prevent large discrepancies that depends on the transmission rather than the storage cost. Read latency was also analyzed and an example of a Smart Contract that was used to test the read latency of *bytes32* can be found in Listing 6.2.

```
pragma solidity ^0.4.4;


contract Bytes32Read {
    bytes32 b;

    function Bytes32Read() public {
        b = 0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef;
    }

    function read() public view returns (bytes32) {
        return (b);
    }
}
```

Listing 6.2: Contract for testing read-latency for the type bytes32

6.1.2 Storage Structures

When we had analyzed the basic types we proceeded by testing the different data structures available in the same manner described in Section 6.1.1. In our tests the storage structures was filled with four instances of data with the same type. The different structures we tested were:

- **Arrays** — Each instance type was stored in normal arrays similar to those that exist in C.
- **Mappings** — For example each uint256 of data was mapped in the mapping type to an uint8 identifier.
- **Parallel** — Each instance of data was stored as a simple variable directly in the global scope.
- **Structs** — A struct was created containing the four instances of the data.

When we tested these data structures we also conducted test to measure the latency of reading the data stored in the structure. This test deployed the contracts containing the storage structures and fetched the data from that instanced contract to get the minimal read latency.

6.1.3 Key Types

Finally we analyzed which cryptographic algorithm was most suitable to use in the DSSO system in the same manner as previous sections. While cost and latency is still crucial, the strength of the algorithm must also be considered. Keys with low cryptographic strengths can potentially be broken given an advanced computer meaning that identities could be compromised. For the ECDSA keys, compressed form was used to decrease the amount of data. All keys that were supported by the Python library we used for cryptography were evaluated [21]. For RSA keys, key-length where chosen so that their strength was comparable to a ECDSA key.

6.2 Authentication

Benchmarks were used for measuring the total authentication time needed by the server. The process consists of the three following steps:

- Fetching the public-key from the contract in the ledger
- Verifying the validity of the message
- Verifying the authenticity of the signature of the message

In the benchmarks for verifying signatures and the message, the Python library *timeit* [20] was used to time the execution. The fetching latency was measured in the Ethereum-specific test regarding read latency. A benchmark of the complete procedure was finally performed to get the latency for the full procedure, using the result of 100,000 iterations. To get an perspective of the speed of this implementation, it was compared to a centralized password-based authentication system implemented in Django. Implementation complexity as well as practical differences, such as space requirements, were also considered as they may affect the usability of this approach compared to a traditional one from a server owner's perspective.

7

Results

In this chapter we present the results from the tests we have conducted to verify the usability and performance of our system.

7.1 Ethereum Storage

This chapter contains results for the tests that have been performed to evaluate the best way to store data in the blockchain. Metrics that have been used to evaluate this are gas cost and read latency. In Section 7.1.1, we present our results regarding the different data types in Solidity, in Section 7.1.2 we present our findings about different data structures and finally in Section 7.1.3 we compare the different key types stored in a manner that has been proven optimal given the results in Section 7.1.1 and Section 7.1.2.

7.1.1 Data Types

In this section the gas cost and latency associated with declaring, assigning and changing basic data types is presented. For gas cost, the result is presented in the gas amount required to execute the operations within the compiled contract. All tests were performed with global declaration and a basic assignment, as in the code example in Listing 6.2. This will from now on be referred to as parallel.

In Table 7.1, we can observe that there are very minor differences in the gas cost for declaring the data types in Solidity. We can also observe that for all types except the dynamic ones, String and Bytes, the cost of assigning data is almost the same regardless of the data size of the variable. The gas cost of assigning 32 bytes is almost the same as assigning 1 byte, it only differs 587 gas or about 0.64% even though 32 times as much data is stored, we therefore choose to explore this further in Table 7.2. The dynamic types Bytes and String diverge from the rest and require approximately 35,000 or 55,000 more gas respectively to store a word of data.

If we compare Table 7.1 with Table 7.2, it can be noted that it is cheaper to use the

Type	Declaration	Assignment	Change	Read Latency (μs)
Bool	68,653	90,560	13,334	5,012.44
Bytes1	68,653	91,097	26,680	4,736.13
Bytes32	68,589	91,684	26,431	4,748.17
Uint256	68,653	91,255	26,414	4,934.61
Uint8	68,525	90,564	26,667	4,827.19
Bytes(array)	68,653	125,083	32,106	6,762.11
String	68,653	146,095	37,194	7,255.62

Table 7.1: Gas cost and read latency associated with the basic data types in solidity

Mismatched types	Declare	Assign	Change	Read Latency (μs)
Bytes32 undersized data	—	89,576	26,431	4,790.68
Uint256 undersized data	—	89,147	26,414	4,960.65
Uint8 oversized data	—	310,583	—	21,064.57
Bool oversized data	—	1,997,961	—	188,483.09

Table 7.2: Gas cost and read latency associated with basic data types used to store mismatched data

bytes327.1 type to store 1 byte of data than to store that data in the bytes1. The same is goes for uint. We also presented results for storing oversized data in several small variables. We chose to explore this for uint since it was both cheaper and faster than bytes1 and for bool that potentially could be used as a bit. Results showed that storing oversized data in smaller variables is not a viable option. Overall, bytes32 and uint256 proved to be the most suitable options given our user-case.

7.1.2 Data Structures

Given that uint256 and bytes32 proved to be most promising in terms of latency and cost per amount of data stored, we used different storage structures to evaluate which one was the most suitable for our purpose, and if data storage affected which one is more advantageous. The benchmarks are presented in Table 7.3 and Table 7.4. From these results we can gather that the most efficient storage structure in terms of both gas cost and read latency is to store the keys in parallel in the global scope, in the majority of the cases.

Structure	Declaration	Assignment	Change	Read Latency (μs)
Array	68,653	189,342	26,703	7,243.28
Mappings	103,490	199,414	26,498	6,692.66
Parallel	68,653	159,253	26,414	6,189.42
Struct	68,653	162,726	26,420	6,133.33

Table 7.3: Gas cost and read latency for different storage structures based on uint256

Structure	Declaration	Assignment	Change	Read Latency (μs)
Array	68,653	192,394	26,720	6,612.06
Mappings	104,708	201,508	26,509	6,025.79
Parallell	68,589	160,973	26,431	5,552.82
Struct	68,653	165,946	26,437	5,589.61

Table 7.4: Gas cost and read latency for different storage structures based on bytes32

7.1.3 Key-Types

The optimal data types and storage structure that was found in Section 7.1.2 and Section 7.1.1. uint256 and bytes32 in parallel storage are used here to benchmark the different algorithms and key sizes in this section. The results show that while bytes32 is always more costly, it can be the faster alternative in some cases. Therefore, none of them can be considered as the optimal choice yet. Note that the verification time is independent on the data-type used since it takes place after it is fetched and formatted correctly.

Since 128 in strength is considered secure using today's technology, this is what we choose to look at in Table 7.6 and Table 7.5. In this category, our results show that secp256r1 and secp256k1 are the cheapest one to store, and that secp256r1 has the lowest total latency for fetching.

Algorithms	Security Strength	Verification Time (μs)	Fetch Latency (μs)	Total Latency (μs)	Deployment Cost (gas)
RSA 1024	80	59.94	5,641.736	5,701.68	159,125
EC sect163r2	80	495.88	4,985.096	5,480.98	90,507
EC sect163k1	80	473.39	4,797.081	5,270.47	90,507
EC secp192r1	96	298.19	5,001.043	5,299.24	90,779
RSA 2048	112	80.28	7,023.217	7,103.50	249,861
EC sect233r1	112	631.20	4,938.751	5,569.95	91,051
EC sect233k1	112	600.07	4,901.14	5,501.21	91,119
EC secp224r1	112	164.08	4,659.245	4,823.33	91,051
RSA 3072	128	112.78	7,694.003	7,806.78	340,461
EC sect283r1	128	1,169.41	5,192.39	6,361.80	112,017
EC sect283k1	128	1,093.69	5,179.177	6,272.86	112,085
EC secp256r1	128	138.57	5,308.145	5,446.71	111,813
EC secp256k1	128	462.35	5,396.295	5,858.65	111,813
RSA 7680	192	409.22	18,966.126	19,375.35	748,257
EC sect409r1	192	1,996.19	5,063.394	7,059.59	113,041
EC sect409k1	192	1,853.90	5,156.207	7,010.11	113,173
EC secp384r1	192	853.92	5,116.718	5,970.64	112,901
RSA 15360	256	1,444.81	33,169.258	34,614.07	1,427,789
EC sect571r1	256	4,620.43	5,409.846	10,030.28	135,023
EC sect571k1	256	4,210.46	5,513.333	9,723.80	134,959
EC secp521r1	256	950.50	5,274.539	6,225.03	134,547

Table 7.5: Benchmarks for uint256

Algorithms	Security Strength	Verification Time (μs)	Fetch Latency (μs)	Total Latency (μs)	Deployment Cost (gas)
RSA 1024	80	59.94	5,728.681	5,788.63	160,845
EC sect163r2	80	495.88	4,621.215	5,117.10	90,872
EC sect163k1	80	473.39	4,487.77	4,961.16	90,936
EC secp192r1	96	298.19	4,743.216	5,041.41	91,208
RSA 2048	112	80.28	6,667.665	6,747.95	253,293
EC sect233r1	112	631.20	4,719.412	5,350.61	91,480
EC sect233k1	112	600.07	4,922.884	5,522.95	91,548
EC secp224r1	112	164.08	4,863.871	5,027.95	91,480
RSA 3072	128	112.78	7,514.18	7,626.96	345,609
EC sect283r1	128	1,169.41	5,068.054	6,237.47	112,875
EC sect283k1	128	1,093.69	4,885.414	5,979.10	112,943
EC secp256r1	128	138.57	5,158.742	5,297.31	112,671
EC secp256k1	128	462.35	4,980.587	5,442.94	112,671
RSA 7680	192	409.22	20,695.609	21,104.83	761,127
EC sect409r1	192	1,996.19	4,986.977	6,983.17	113,899
EC sect409k1	192	1,853.90	4,949.167	6,803.07	114,032
EC secp384r1	192	853.92	5,084.688	5,938.61	113,759
RSA 15360	256	1,444.81	36,552.681	37,997.49	1,453,529
EC sect571r1	256	4,620.43	5,152.182	9,772.61	136,310
EC sect571k1	256	4,210.46	5,272.703	9,483.17	136,246
EC secp521r1	256	950.50	5,473.122	6,423.62	135,834

Table 7.6: Benchmarks for bytes32

7.2 Full Authentication Process

For a server to be able to use DSSO, it needs to be able to access an Ethereum node. This differs from existing methods that only needs access to a database and possibly libraries implementing the authentication process. The blockchain for a full Ethereum node currently requires approximately 300GB [9] of data while the size of our Django database used for testing only requires a couple of hundred kilobytes.

Table 7.7 shows that authenticating using DSSO is quite a bit faster than the traditional one, regardless of which data-type is used to store the key. Due to the fluctuations in authentication time we have chosen to also present the latency in histograms which can be found in Figure 7.1 - 7.3.

Type	Minimum (ms)	Mean (ms)	Median (ms)	Maximum (ms)
DSSO bytes32	17.94	29.04	28.80	49.66
DSSO uint256	18.12	28.90	28.65	57.14
Traditional	75.18	79.22	77.61	98.18

Table 7.7: Authentication latency measured in milliseconds

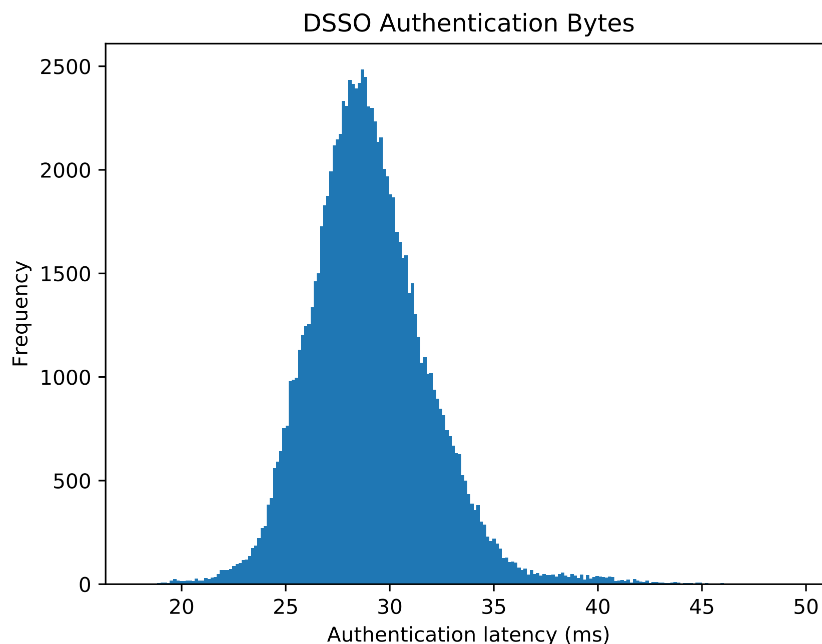


Figure 7.1: The authentication delay for the final contract that uses the bytes32 type for storage

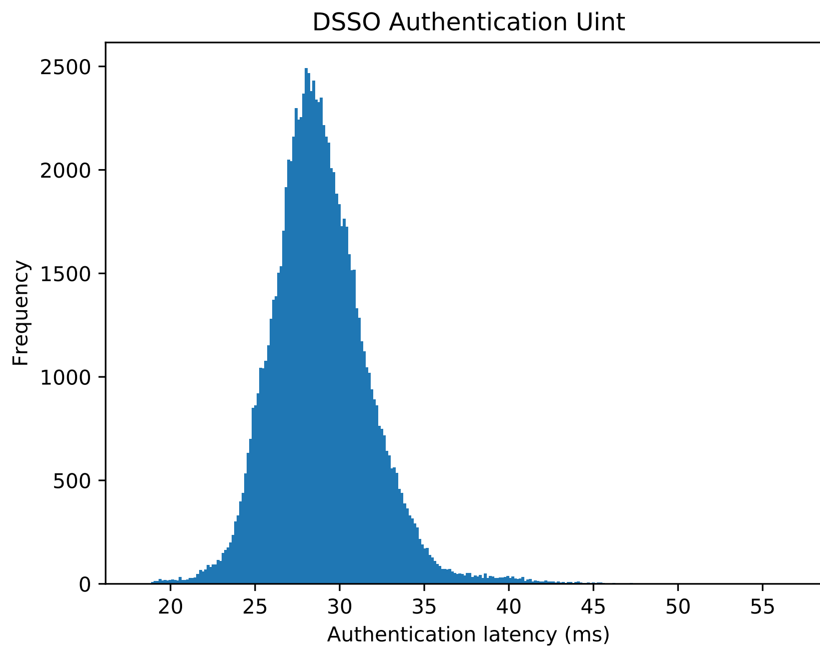


Figure 7.2: The authentication delay for the final contract that uses the uint256 type for storage

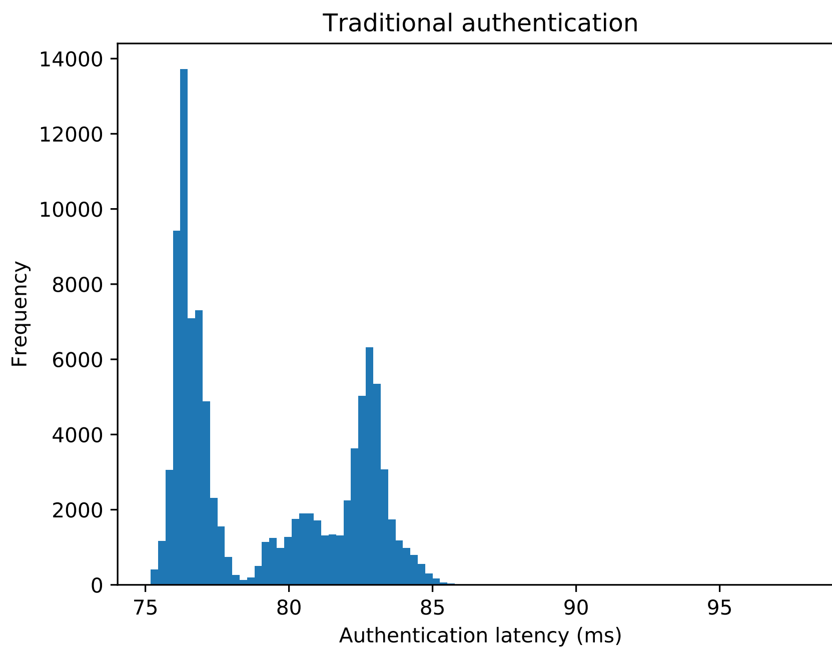


Figure 7.3: The authentication delay for the traditional authentication system

8

Discussion

In this chapter we discuss the outcomes of our evaluation, interesting observations and factors that could have affected the results. The usability of the system will be discussed, as well as comparisons with existing solutions.

8.1 Client

Storing a key in Ethereum, using the cheapest method with the datatype uint256 costs 111,813 gas which can be bought for between $0.148 and $1.478 depending (at the time of writing) on how long you are willing to wait for the block to be confirmed (71 seconds and 31 seconds respectively). The data is based on information for block 5,502,133 and could have changed since [6]. The cost of storing one's public key in the blockchain could therefore be considered as reasonably low considering that costs less than a cup of coffee. This is also true if one wishes to utilize a key with 256 bits of strength ($0.178—$1.779).

Based on our results in Table 7.7 users will not suffer from higher latency when logging in with DSSO compared to logging in with username and password. Instead, registering oneself will be the tricky part. Since distributed consensus only reaches consensus eventually, actors that handle transactions does not consider freshly mined blocks as valid. Only when a certain amount of blocks has been confirmed after it, will they consider it as valid since the possibility of forks decreases with the number of confirmed blocks after it. Companies that offer transactions involving Ether consider a transaction to be valid after 30—100 block confirmations [12, 8, 4]. Services providing authentication would probably want to handle forks similarly. Therefore, one would have to wait between 15—118 minutes, depending on previously mentioned factors in form of gas cost and confirmed blocks, before using the deployed contract to log in for the first time. Assuming that the same contract is used for several or all services, this latency could be considered as tolerable for some users.

8.2 Server

Latency of the authentication process is crucial for the server for two reasons. Firstly, the scalability of the implementation is highly dependent on it since more authentications per time unit means that more users can be served in a certain amount of time. The more time it takes to serve a user, the more likely it is that the user will end up avoiding using your service. Secondly, the less time it needs to spend on authenticating users, the more time it can spend on other valuable problems that need computation time.

Benchmarks showed that the algorithm secp256r1 was the fastest overall when it comes to authentication, and therefore results in a quicker authentication overall. Server owners would therefore prefer to use it. As this also generated the cheapest keys to store, users would probably not mind using it either. Stronger keys can be used in for a slightly higher cost but can generate scalability issues on the server-side as secp521r1 for example takes approximately six times longer to verify.

As mentioned earlier, compared to a server utilizing Django's traditional method of authentication the DSSO latency is actually faster. This might be caused by the fact that Django uses 100,000 iterations of the pbkdf2 hashing algorithm to secure the stored password [18]. The low latency of DSSO means that it can be considered as a viable method of authentication but since we have not evaluated how the latency is affected by higher computational loads or concurrency, we can only reason about the scalability under circumstances where each authentication takes a constant amount of time. Under these circumstances, the DSSO method of authentication can be considered scalable in regard to latency. Other factors that could have impacted the latency is the blockchain used during testing. It was a test network that was not part of the main Ethereum network. Both the size of the blockchain and the traffic caused by other transactions did not exist, which could have lowered the latency.

The space requirements for using the main network may cause problems for more lightweight servers that wants to implement the DSSO, however the Ethereum node can easily be moved to a remote server in exchange for a slightly higher fetching latency. Regarding the Ethereum node, it can in theory run on any computer in the world as long as it accepts Remote Procedure Calls (RPC). But from a security perspective, the info received from a node is only as trustworthy as the computer it runs on. In practice it means that the most secure option is to fetch information from a node run locally, and that the least secure option is to fetch information from an arbitrary node in another network.

8.3 Security

Since our concept is highly reliant on cryptography and cryptocurrency, it naturally follows that it inherits the security flaws that are associated with those techniques.

The potential attack vectors that are added in the DSSO approach are ones related to the registration stage and the login stage. The signed message has to safely make its way from the storage in the web extension to the service provider for an authentication to be successful, and there are things that can go wrong along the way. If the signed message is retrieved by an attacker using eavesdropping or some type of phishing within the 30 second window an impersonation attack can take place. This can be compared with tokens that are currently used in SSO solutions, which also can be used to impersonate users.

9 Conclusion

We have presented a proof-of-concept, utilizing a distributed way of storing identities in a PKI-like manner, of authenticating users towards services. The security and usability relies on the implementation as well as the cryptocurrency chosen as underlying storage for the identities. We have implemented a system, based on the Ethereum technology, that is feasible to use both from a user's and a server owner's perspective. The user can expect faster authentications than traditional ones. But registration to the network is slower since the contracts need to be added to the blockchain, but this is not a large problem due to the fact that this only needs to be done once. When registering to a service the authentication will be conducted in the same manner as an ordinary sign-on and should therefore be fast. The cost for the registration to the network is of course problematic since similar single-sign on solutions offered by for example Google and Facebook are free to use. If the user is willing to pay for slightly higher security, longer keys with higher strength can be stored in the blockchain. Keys can otherwise be switched by deploying a new contract later if needed. Considering today's price of Ethereum, the cost of registration could be compared to the price of a cup of coffee, which means that it is affordable for most people.

It is complicated to reason about whether it is worth the cost from a user's perspective since everyone values privacy differently and there are no existing services similar to this, to our knowledge, that charges for providing an identity. A server owner can expect a somehow scalable solution, assuming that there are resources for running the Ethereum node, that can service clients with low latency. Although it has similar properties as for example an SQL database, it does require both more space and more computation power. The scalability of this approach has yet to be explored. The authentication has a latency in the same magnitude as the traditional ones.

9.1 Future Work

The performance on large scale systems should be evaluated for DSSO to be deemed as a total viable option compared to the larger actors such as Google and Facebook.

Testing on systems on such scales was not possible during this thesis, thus we have only provided a theoretical scenario for this usage.

In the results we presented the costs for changing data in the contracts. This sort of functionality can be useful if users for any reason wants to switch their keypair. This is particularly useful if the keypair has been jeopardized or if a stronger cryptography algorithm is wanted. Solidity also enables functions to be written that can only be called by the owner of the contract. While this does introduce potential risks if the private key gets revealed, it can serve as an extra security layer assuming that the private key is kept safely. Similar to the SSL certificates, the algorithm that is used can also be specified in the contract, to ensure compatibility with multiple alternatives. These functions and many more can help increase the security and usability even further in the DSSO system since Solidity is Turing-complete.

Utilizing other blockchain technologies, either an existing or a custom-made one, can also increase the performance for the user. Although Ethereum has not been proven to be a particular bad choice for these purpose, it is not a technology optimized with regard to identity storage. Blockchain technologies aimed towards storage in specific would be interesting to evaluate. The storage requirements could be decreased since ordering, that is part of the Ethereum technology, is not of great importance for this purpose since the deployed identities have no causal relations between each other. All transactions that have absolutely nothing to do with DSSO, but have to be stored anyways, could also be removed if there was a blockchain specifically designed for storing identities.

Bibliography

[1] Ethereum: White paper. https://github.com/ethereum/wiki/wiki/White-Paper/, 2016. [Online; accessed 26-January-2018].

[2] Solidity. https://solidity.readthedocs.io/en/latest/, 2017. [Online; accessed 1-March-2018].

[3] Cryptocurrency market capitlizations. https://coinmarketcap.com/, 2018. [Online; accessed 1-March-2018].

[4] Deposit does not arrive. https://support.binance.com/hc/en-us/articles/115003736451-Deposit-does-not-arrive, 2018. [Online; accessed 25-April-2018].

[5] Emercoin. https://emercoin.com/en, 2018. [Online; accessed 01-May-2018].

[6] Eth gas station. https://ethgasstation.info/, 2018. [Online; accessed 25-April-2018].

[7] Ether. http://www.ethdocs.org/en/latest/ether.html, 2018. [Online; accessed 17-April-2018].

[8] Ether - sending and receiving. https://support.zebpay.com/hc/en-us/articles/360000265209-Ether-Sending-and-Receiving, 2018. [Online; accessed 25-April-2018].

[9] Ethereum (eth) price stats and information. https://bitinfocharts.com/ethereum/, 2018. [Online; accessed 19-March-2018].

[10] Ethereum: White paper. https://github.com/ethereum/wiki/wiki/White-Paper, 2018. [Online; accessed 1-March-2018].

[11] Ganache. http://truffleframework.com/ganache/, 2018. [Online; accessed 27-February-2018].

[12] How long does it take to receive bitcoin or ethereum? https://www.luno.com/help/en/articles/11000033687-how-long-does-it-take-to-receive-bitcoin-or-ethereum, 2018. [Online; accessed 25-April-2018].

[13] Introduction to smart contracts. https://solidity.readthedocs.io/en/v0.

4.22/introduction-to-smart-contracts.html, 2018. [Online; accessed 17-April-2018].

[14] jsrsasign. https://github.com/kjur/jsrsasign, 2018. [Online; accessed 26-April-2018].

[15] Metamask: How it works. https://metamask.io/, 2018. [Online; accessed 27-February-2018].

[16] Node.js v10.0.0 documentation. https://nodejs.org/api/process.html, 2018. [Online; accessed 30-April-2018].

[17] Public key infrastructure. https://msdn.microsoft.com/en-us/library/windows/desktop/bb427432%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396, 2018. [Online; accessed 17-April-2018].

[18] Source code for django.contrib.auth.hashers. https://docs.djangoproject.com/en/2.0/_modules/django/contrib/auth/hashers/, 2018. [Online; accessed 07-May-2018].

[19] Structure of a contract. http://solidity.readthedocs.io/en/v0.4.21/structure-of-a-contract.html, 2018. [Online; accessed 17-April-2018].

[20] timeit — measure execution time of small code snippets. https://docs.python.org/3.6/library/timeit.html, 2018. [Online; accessed 16-March-2018].

[21] Welcome to pyca/cryptography. https://cryptography.io, 2018. [Online; accessed 9-April-2018].

[22] What is an ssl certificate? https://www.globalsign.com/en/ssl-information-center/what-is-an-ssl-certificate/, 2018. [Online; accessed 01-May-2018].

[23] Daniel R. L. Brown. Standards for Efficient Cryptography SEC 2 : Recommended Elliptic Curve Domain Parameters. 2(Sec 2), 2010.

[24] Ronald H Brown. FIPS PUB 180-1 - FEDERAL INFORMATION PROCESSING. pages 1–23, 1995.

[25] Quynh H. Dang. Secure Hash Standard. *FIPS*, (August), 2015.

[26] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. Technical report, Cambridge, aug 2008.

[27] W. Diffie and M. E. Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 67(3):397–427, March 1979.

[28] Communications Security Establishment. Implementation Guidance for FIPS PUB 140-2 and the Cryptographic Module Validation Program. *Management*, pages 1–63, 2006.

[29] European Union Agency for Network and Information Security. *Algorithms, key size and parameters report - 2014*. 2014.

[30] Jim Foti. FIPS PUB 202 SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions CATEGORY: COMPUTER SECURITY SUB-CATEGORY: CRYPTOGRAPHY. 2015.

[31] Django Software Foundation and individual contributors. Django at a glance. `https://docs.djangoproject.com/en/2.0/intro/overview/`, 2018. [Online; accessed 27-February-2018].

[32] Google. Extensions: Overview. `https://developer.chrome.com/extensions/overview`, 2018. [Online; accessed 26-February-2018].

[33] Emercoin International Development Group. Emcssl - decentralized identity management, passwordless logins, and client ssl certificates using emercoin nvs. 05 2015.

[34] AgileBits Inc. Security. `https://1password.com/security/`, 2017. [Online; accessed 26-January-2018].

[35] LogmeIn Inc. How it works. `https://www.lastpass.com/how-lastpass-works`, 2018. [Online; accessed 26-January-2018].

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

[37] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[38] Y. Liu, R. Li, X. Liu, J. Wang, L. Zhang, C. Tang, and H. Kang. An efficient method to enhance bitcoin wallet security. In *2017 11th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, pages 26–29, Oct 2017.

[39] Christian Lundkvist, Rouven Heck, Joel Torstensson, Zac Mitton, and Michael Sena. Uport: A platform for self-sovereign identity, 2016.

[40] Mozilla and individual contributors. What is a web server? `https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_web_server`, 2018. [Online; accessed 26-February-2018].

[41] U. Mukhopadhyay, A. Skjellum, O. Hambolu, J. Oakley, L. Yu, and R. Brooks. A brief survey of cryptocurrency systems. In *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, pages 745–752, Dec 2016.

[42] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 03 2009.

[43] Priyadarshini Patil, Prashant Narayankar, D. G. Narayan, and S. M. Meena. A Comprehensive Evaluation of Cryptographic Algorithms: DES, 3DES, AES, RSA and Blowfish. *Procedia Computer Science*, 78(December 2015):617–624, 2016.

[44] Ronald Rivest. RFC 1186 - MD4 Message Digest Algorithm. 1990.

[45] Ronald Rivest. RFC 1321 - The MD5 Message-Digest Algorithm. 1992.

[46] Marc Stevens, Pierre Karpman, and Thomas Peyrin. Freestart collision for 76-step SHA-1 Freestart attack. 2012:1–21, 2014.

[47] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[48] A. Takeda, K. Hashimoto, G. Kitagata, S. M. S. Zabir, T. Kinoshita, and N. Shiratori. A new authentication method with distributed hash table for p2p network. In *22nd International Conference on Advanced Information Networking and Applications - Workshops (aina workshops 2008)*, pages 483–488, March 2008.

[49] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. (90304009):17–36, 2005.

[50] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. 2005.