



CHALMERS

Secure Cloud Storage

B.Sc. Thesis in Computer Science and Engineering

Ahlstedt, Mattias
Altensten, Simon
Erlandsson, Andréas
Hildén, Johannes
Johansson, Rasmus
Reitmaier, Rebecka

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden 2017

Title: Secure Cloud Storage

Authors: Mattias Ahlstedt, Simon Altensten, Andréas Erlandsson,
Johannes Hildén, Rasmus Johansson, Rebecka Reitmaier

Supervisor: Michal Palka

Examiner: Thomas Hallgren

- © Mattias Ahlstedt, 2017
- © Simon Altensten, 2017
- © Andréas Erlandsson, 2017
- © Johannes Hildén, 2017
- © Rasmus Johansson, 2017
- © Rebecka Reitmaier, 2017

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96
Gothenburg, Sweden
Telephone + 46 (0)31 - 772 1000

The Authors grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Abstract

This bachelor of science thesis describes and discusses the development of a secure cloud storage service. The service is secure in the sense that no one other than the owner of the file can access the information stored with our service. Our solution does not require implementation of a cloud storage service but instead uses an existing cloud service and adds a security layer to it. On top of that we have also produced a protocol for how to share encrypted files between users of the system without revealing any vital information to the server, nor to any potential adversaries.

Keywords: End-to-end Encryption, E2EE, Client-side Encryption, Secure Cloud Storage

Sammanfattning

Detta kandidatarbete beskriver och diskuterar utvecklingen av en säker molnlagringstjänst. Säker syftar här på att endast ägaren av en fil kan komma åt informationen som lagrats. Vår lösning lägger till funktionaliteten som säkrar informationen på existerande molntjänster för lagring, vilket innebär att vi inte behöver implementera en helt ny molntjänst från grunden. Utöver detta har vi även skapat ett protokoll för att kunna dela filer mellan användare av tjänsten utan att exponera informationen för varken servern eller andra obehöriga.

Acknowledgements

We would like to thank our supervisor Michal Palka for his help throughout the entirety of the project. Michal has guided us through both technical and non-technical problems. He has also broadened the group's view which has led to a more interesting project.

Contents

1	Introduction	1
1.1	Background	1
1.2	Purpose	2
1.3	Problem statement	2
1.3.1	Client-side encryption	2
1.3.2	Account management	2
1.3.3	Cloud storage	2
1.3.4	Sharing of files	3
1.4	Scope	3
1.4.1	Application type	3
1.4.2	Cryptographic libraries	3
1.4.3	Handling files	3
1.4.4	Client encryption key	4
1.4.5	Cloud service	4
1.4.6	Target audience	4
1.4.7	Smartphone compatibility	4
2	Theory	5
2.1	Symmetric encryption	5
2.1.1	Block ciphers	5
2.1.2	Block cipher modes	6
2.2	Asymmetric encryption	6

2.3	Hashing	7
2.4	Pseudo random number generation	8
2.5	Key derivation function	8
3	Method	9
3.1	Early stages of the project	9
3.2	Implementation	9
3.3	A tool for privacy	9
3.4	Work methodology and tools	10
4	Result	11
4.1	Graphical user interface	11
4.2	High-level architecture	13
4.3	How security is achieved	13
4.3.1	Registration	15
4.3.2	Login	16
4.3.3	Uploading a file	17
4.3.4	Downloading a file	18
4.3.5	Sharing a file	19
4.3.6	Changing passphrase	21
4.4	Front-end	21
4.4.1	How Elm was chosen	21
4.4.2	The Elm architecture	23
4.4.3	Cryptography in the browser	24

4.4.4	Algorithm choices	25
4.4.5	Choice of block cipher mode of operation	26
4.4.6	File manipulation	26
4.5	Back-end	26
4.5.1	Python - Back-end language	27
4.5.2	Tornado - Communication between front-end and back-end	27
4.5.3	Database	27
4.6	Cloud Storage	28
5	Discussion	30
5.1	Security considerations	30
5.2	Security vulnerabilities	31
5.3	Forward secrecy	31
5.4	Scalability	32
5.5	Evaluation of front-end	32
5.6	Evaluation of back-end	33
5.7	Design principle and architecture of the back-end	33
5.8	Related work	34
5.9	Haste.App	34
5.9.1	Choosing Haste.App	34
5.9.2	Moving away from Haste.App	34
5.10	Ethical perspectives	35
6	Conclusion	36

1 Introduction

This section provides a background to the project and motivates why it is of interest. It also defines and clarifies the main problem and the scope of the project.

1.1 Background

The emergence of the Internet had a huge impact on the way we share and store information. Even though it started off small it has now evolved into something that many people are depending on in their everyday life. The evolution of the Internet has introduced many new concepts, amongst them is cloud storage. Cloud storage refers to storing data remotely through the use of the Internet, often on servers hosted by big companies such as Google and Amazon, thereby making it accessible to you wherever you are.

Even though cloud storage is very convenient and makes plenty of people's work easier, you have to consider who has access to the things you store there. In most cases when you store information in the cloud through a third-party provider, the information is encrypted on their servers to prevent anyone but you from accessing your information. There are two problems with that they encrypt the data themselves. The first problem is that the cloud service provider has access to both the encryption keys and the encrypted files. This means that they have full access to anything you store at their servers. The second problem occurs if the cloud provider gets hacked, then the hacker has full access to all the information since both the encrypted files and the keys are stored in the same place. A solution to both of these problems is to use client-side encryption [1].

When using client-side encryption, the encrypted data is still stored with the cloud providing company. The major difference is that the encryption keys are kept by the user, ensuring that only the user has access to the information [1]. This means that the information is unreadable to both the cloud providing company as well as an adversary.

Today there are several big cloud service providers with well developed infrastructure, such as Google Drive and Dropbox [2][3]. However, both of these lack the client-side encryption. Then there are several smaller companies that provide client-side encryption but fail to deliver on the infrastructure, for example Sync and Crypho [4][5].

Our vision for this project is to build a service that combines the powerful infrastructure of giants such as Google Drive or Dropbox with the security and privacy of client-side encryption. Our biggest competitor that manages to combine these two is called Tresorit [6].

1.2 Purpose

The purpose of the project is to develop a web-based secure file storage and sharing service targeted at companies. It is secure in the sense that only the owner of a file and users who have had the file shared with them may access the information. This implies that the service provider has no knowledge of what information is stored with their service.

1.3 Problem statement

The main challenge of the project is to provide absolute privacy. No one but the people you choose should have access to your information. The following subproblems are identified as crucial parts of the project.

1.3.1 Client-side encryption

In order to ensure that the stored data is accessible only by the owner of the file, or by any accounts that have been granted permission by the owner, all encryption needs to be handled client-side. This means that the server is never to have access to any of the users' cryptographic keys. A way to handle metadata, file storage and key storage is necessary to ensure this.

Since the solution is supposed to be web-based, performance and security properties will be taken into consideration when choosing and integrating the necessary cryptographic protocols that are to be run in a web browser.

1.3.2 Account management

Each user should be able to register with an email and a passphrase. A passphrase is a longer password that may include blank spaces. When choosing a passphrase, restrictions which guarantees the strength of the passphrase will be needed.

There is also need for a way to handle the case of a user unregistering from the system. Any future versions of any files that were shared with the previous user should no longer be accessible for them.

1.3.3 Cloud storage

Our service will be compatible with a variety of different external cloud services. This is to ensure that it is possible for us to offer companies the option to choose what cloud

service they want to use. Therefore, all parts of the application that interacts with the cloud storage must be developed in a way that facilitates modification.

1.3.4 Sharing of files

A major concern with the system is how to handle the sharing of files between users. To allow users to share the encrypted data amongst one another, the keys for the encrypted files must be passed around between users. A protocol for handling sharing that denies the server and potential adversaries access must therefore be devised.

1.4 Scope

Due to the constrained time frame of the project, some limitations regarding the functionality of the end product are set.

1.4.1 Application type

The service is going to be a web application which consists of both a client-side component and a server-side component. Alternatives to a web application would have been to develop a native application or a browser extension. The reason for choosing a web application is that it is cross-platform and cross-device by default, as opposed to a native application or a browser extension. A native application or a browser extension would have needed extra work to be functional on for example both a computer running windows and a phone running android. Another advantage of a web application is that it does not require installation.

1.4.2 Cryptographic libraries

None of the cryptographic algorithms that are necessary for the service to function will be implemented by us. Existing cryptographic libraries will be used.

1.4.3 Handling files

The service only support the downloading, uploading and sharing of files. There will not be an implementation of live editing. If a file is to be edited, it has to be downloaded, locally edited, and then uploaded again with the same settings as the previous file.

1.4.4 Client encryption key

The system is not going to use any hardware based keys. The only thing a user needs to access their account is their passphrase. A hardware key is a key file which is stored in a specific unit's hardware. The reasons for not using a hardware key is that it complicates accessing your account from multiple devices. Also, it is troublesome to access files on the storage of the local machine from the browser without requiring user interaction.

1.4.5 Cloud service

Instead of developing directly against an existing cloud storage company, a server with an open source based cloud storage solution will be used. This is to simplify the development process. A future migration to another cloud service will be possible but is not part of the scope of this project.

1.4.6 Target audience

The product that is being developed is deemed suitable for companies since it is common for them to have large volumes of confidential information. Because of this and in order to narrow down the project and get more strict user needs, the target audience will be set to companies.

1.4.7 Smartphone compatibility

The system's graphical user interface (GUI) does not have a focus on being compatible with smartphones or tablets.

2 Theory

Information security commonly mentions three properties which needs to be fulfilled in order to communicate securely: confidentiality, integrity and authenticity [7]. All of these properties are most often achieved through the use of cryptography.

Confidentiality means that none other than the sender and intended recipient should be able to understand the information. Integrity means that the receiver can verify that the received information is identical to the sent information. Lastly, authenticity means that the recipient knows that the sender is who they claims to be [7].

This section gives a presentation of the cryptographic theory and some other technologies and concepts that are required to fully understand this thesis. It will present the cryptographic primitives used in this project as well as give an overview of how they work.

2.1 Symmetric encryption

Symmetric encryption is probably the most fundamental primitive used in any cryptographic system. It is the simple principle of locking information with one key. Symmetric encryption uses a key k and a pair of an encryption and a decryption algorithm (E, D) . To encrypt a plaintext p you would need to use the key and the encryption algorithm to produce the ciphertext $E(k, p) = c$. To retrieve the plaintext from the ciphertext you would need to use the decryption algorithm D and the same key as before $D(k, c) = m$ [8].

There are two types of symmetric-key encryption, one is called block ciphers and the other one is called stream ciphers [9]. In this bachelor thesis only block ciphers are used and the focus will therefore be on explaining those.

2.1.1 Block ciphers

When you have a fixed-size block of data, for example a 128-bit block, you can encrypt it using a block cipher. Block ciphers encrypt a fixed-size block of plaintext data and outputs a ciphertext of the same size. A block cipher utilises a key k which is used with the encryption algorithm E , $E(k, p) = c$ where c is the result ciphertext and p is the plaintext. The same applies for the decryption algorithm D , $D(k, c) = p$.

There are many block ciphers available, but not all of them are considered good. Therefore the U.S. government decided to create a standard for symmetric encryption. The current standard is called "Advanced Encryption Standard" (AES). The U.S. government did not want to create this standard by themselves, therefore they started a competition in which researches could submit their proposals. Out of five finalists the Rijndael encryption

algorithm won and have since 2001 been the AES. Common key-sizes for AES are 128, 192 and 256 bits [8].

2.1.2 Block cipher modes

Since a block cipher can only encrypt one block of plaintext, there is need for a way to encrypt multiple blocks using a block cipher algorithm and the solution is block cipher modes. A block cipher mode is a protocol for how to use a block cipher to encrypt plaintext of a size greater than one block and the greater part of the block cipher modes requires that the length of the plaintext is a multiple of the block size [8].

Two of the most used block cipher modes are *CBC* (cipher block chaining) and *CTR* (counter). *CBC* is a block cipher mode where the output of the previous block is used to generate the output of the next block. This is done in order to ensure that two equal plaintext blocks never output the same ciphertext blocks. This is done by manipulating the current block of plaintext with a bitwise XOR operation together with the previously encrypted block of ciphertext before encrypting it. This is the definition of the ciphertext for block i [8]:

$$C_i = E(K, P_i \oplus C_{i-1})$$

Since every block depends on the previously encrypted block, the very first encryption of this block cipher mode requires some additional input. This input, C_0 , the so called initialisation vector or *IV*, has to be chosen randomly if the first block is to be encrypted in a non predictable way [8].

CTR is a block cipher mode that produces a stream cipher. A stream cipher does not use the plaintext as an input for creating the ciphertext as *CBC* does. Instead the cipher creates a pseudorandom stream of bytes, also called key stream. The plaintext is then manipulated with a bitwise XOR operation with this generated key. A nonce is used for creating randomness in the *CTR* mode so that the same plaintext never generate the same ciphertext. A *nonce* is a number which is only used once and is always unique. *CTR* is defined below, where K_i is the key stream [8]:

$$K_i = E(K, Nonce||i)$$

$$C_i = P_i \oplus K_i$$

2.2 Asymmetric encryption

Asymmetric encryption is a family of encryption algorithms where you use a key pair instead of a single key. The algorithms usually consists of a secret key sk , a public key pk , an encryption algorithm E and a decryption algorithm D [10]. The major difference in comparison to symmetric encryption is that one key is used for encryption and the other for decryption.

Let us consider the use of such an algorithm where you want to send the message m to a recipient. Before any messages can be sent there is some necessary setup. The receiver must generate a key pair (pk, sk) and publish the public key [11]. The sender can then encrypt the message with the public key of the recipient and produce the ciphertext $E(pk, m) = c$. For the receiver to read the plaintext message m they simply needs to decrypt the ciphertext with their secret key, $D(sk, c) = m$.

The security of these algorithms is based on an assumption of computational hardness [11]. This means that the security is entirely based on the premise that it is computationally unfeasible to solve the problem without knowing the secret key. For example, if you take two large prime numbers p_1 and p_2 and multiply them to get the product $p_1 * p_2 = n$. The operation is easy to perform, but it is deemed unfeasible to find the primes p_1 and p_2 , if they are sufficiently large, given only the product n . So if encryption is based on using the product n and decryption is based on using the primes p_1 and p_2 , then it is easy to send ciphertexts but hard to generate plaintexts from these ciphertexts without knowing p_1 and p_2 .

Asymmetric encryption algorithms are performing slower compared to symmetric encryption algorithms and are therefore mostly used for exchanging or agreeing upon a key that can be used for symmetric encryption [11].

Some of the asymmetric encryption algorithms have a secondary use, ensuring authenticity [10]. This is due to the fact that some of these algorithms works both ways. A sender can encrypt their name with the secret key and produce a signature $E(sk, m) = \sigma$. The signature can then be appended to a message and verified by the recipient by performing the following calculations $D(pk, \sigma) = m$, which yields the original message. The public key pk is published as of the setup and therefore anyone can verify the author of the message. For performance reasons it is commonly the hash of the message that is signed, i.e. $E(sk, H(m)) = \sigma$, where H is a cryptographic hash function.

2.3 Hashing

Cryptographic hash functions are deceptive in that the name is easily confused with the hashmap. This report is only concerned with cryptographic hash functions and any mention of hash functions will therefore refer to the cryptographic ones.

A hash function is a function which scrambles the input into something that seems to be completely random data. The cryptographically secure hash functions are also deterministic one-way functions which means that it is computationally unfeasible to determine the input given only the output. The properties of hash functions makes them suitable for storing login information on servers. Instead of storing the password on the server, you can store a hashed version of the password. Later at login, you can hash the password client-side, send it to the server and then compare to the stored hash on the server. This way the server can be sure that the user entered the correct credentials, without knowing

the actual credentials.

An important property of hash functions is collision-resistance. Since there is an infinite number of inputs but only a finite number of outputs there will be collisions. The collision-resistance property does not eliminate collisions, it merely declares that these collisions can not occur in a feasible time frame [8].

2.4 Pseudo random number generation

A computer is a deterministic machine which implies that it is incapable of coming up with completely unpredictable numbers [8]. To solve this problem humans have come up with a way for the computer to construct numbers that seem random. To do this, the computer takes external inputs from the outside world and then does certain computations on those inputs [8]. The inputs could for example be timings of mouse movements or keyboard strokes, making the numbers seem random [12]. The problem is that if you for two different occasions happen to have the same input data you will get the same output number, thereby only making the output number pseudo random. The computation of calculating the random inputs into a number is done by complex algorithms. These complex algorithms are usually hash functions [13].

2.5 Key derivation function

A key derivation function (KDF) is used to derive cryptographically secure keys from arbitrary inputs that lack the properties of cryptographically secure keys [14]. Along with this input, the function also uses a cryptographic salt to derive the new cryptographic key. A salt is some data which gives the KDF resistance against so called dictionary attacks [15]. The KDF can be even harder to crack by iterating the function several times, at the cost of time efficiency. A KDF can for example be used for creating a cryptographic key from an arbitrary phrase which is easy to memorise for a human.

3 Method

Understanding and solving problems is something that can be done very differently. Throughout this project the group has gone through many different stages to solve the problems connected to the task of creating a secure cloud storage service. Below is a description of how the group has worked throughout different parts of the project.

3.1 Early stages of the project

In the beginning of the project, most of the time was spent on research. The entire group started off reading about the design principles and practical applications of cryptography. The cryptography theory is the most important and complicated part of the project. It is also something that was completely new to most of the group members in the beginning of the project. After obtaining a good basic understanding of the cryptography theory the group moved on to researching how to build a complete system consisting of both a front-end and a back-end and what parts to include in the two. The group also researched what development tools and languages were required for developing the desired product.

To minimise the time spent on research we made sure to always split up the reading tasks amongst the entire group. Each person then summarised the reading into a document which then was shared with the rest of the group. The group could then read it and get a quick overview, thereby saving time and effort. We also made sure to spend a small amount of time discussing each persons summary in a meeting to make sure that the entire group had the same level of understanding.

3.2 Implementation

When the development environment was set up and ready the group started with creating prototype programs. One prototype program was developed for every individual feature to get a better understanding of how difficult it would be to implement. Examples of such features were encrypting a file, downloading a file, uploading a file and keeping track of user accounts. After making sure that all of the core functionality was possible to implement we started putting the prototypes together and working on what would eventually become our prototype.

3.3 A tool for privacy

When talking about privacy on the Internet there really is only one solution, cryptography. Since the cryptography area is very well developed the group did not see any advantages

for creating their own cryptography protocols. Because of this the group only has used existing cryptography protocols from well known libraries.

When creating the system's first security protocol the group did it through an extensive discussion. In this discussion the group talked about many different aspects as efficiency and privacy. The most important aspect was the security of the application. The system's protocol went through many iterations where the group discussed aspects as possible attacks and flaws.

3.4 Work methodology and tools

With six members in the group some coordination was needed. To structure the work during the project a version of the popular agile management process Scrum has been used[16]. The work has been split up into sprints of two weeks and in the beginning of every sprint there was a meeting for planning the upcoming sprint, deciding what tasks to include, how long each task should take to complete and who should be in charge of each task. This helped with making sure that work got done and was done in a organised and efficient manner. The usage of Scrum also made sure that no unnecessary or duplicated work was done. In the end of every sprint there was a meeting for reviewing the latest sprint, talking about what went well and what could be improved for the next sprint. Our supervisor has helped us review and plan our sprints, he also acted as the product owner. To help keep track of the sprint tasks the collaboration tool Trello was used[17]. In Trello the tasks was defined and marked with how much time they would take. The usage of Scrum and Trello gave a good overview of who did what tasks and that no work was being forgotten.

The group has also had a lot of work sessions were the group met and worked together. In these meetings we worked together or by ourselves on different tasks. We believe that these meetings were very efficient since if a question arose it could quickly be discussed and solved together.

For communication the group used Slack [18]. Slack is a communication app where you can have different chat channels within a group. The group used the different channels depending on the subject, for example the group had a "general" channel and a "meetings" channel. The different channels made it easier to find information within Slack.

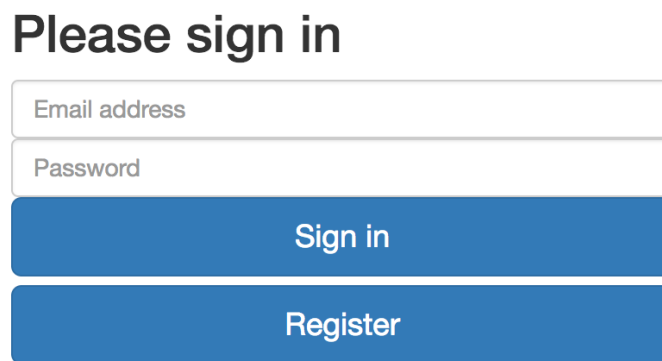
Throughout the project Git [19] has been used for version controlling the code. Along with Git have we used the web service Github [20] to store the code online and share it between the group members.

4 Result

This section present how the application looks and works. Planned functionality that has not yet been implemented will also be presented, but as how it is intended to work in the future. This unimplemented functionality will be described with a clear distinction so that there is no doubt in what is implemented and what is not.

4.1 Graphical user interface

When the user first encounters the application they start at the login page where they either can sign in or register to the system, Figure 1 shows the GUI. If the user tries to sign in by providing a correct email address and passphrase, they will be logged in and forwarded to the main page. If the email address or passphrase is wrong, a small red text will be shown with the text "wrong" and the view will not change.



The image shows a login form with the title "Please sign in". It contains two input fields: "Email address" and "Password". Below the input fields are two blue buttons: "Sign in" and "Register".

Figure 1: The start page of the application.

If the user clicks on the "Register" button the system will forward the user to a registration view, Figure 2 shows the GUI for registering. When the user has registered they are directly forwarded to the main page.



Figure 2: The register page of the application.

The main page is where you access the core functionality of the product, the GUI of the main page is shown in Figure 3. Since the idea was to have teams with shared files, the GUI has been influenced by it even though some functionality has not yet been implemented. To the left in Figure 3 is the team-section. In the team-section the teams associated with the user are supposed to appear. However, teams as a concept is dependant on the sharing functionality, and since sharing has not been implemented neither has the teams. If the user clicks on a team, the file-section would have updated and shown the files which was shared within that team. The button "Manage Team" was supposed to change the view into a new view where the user could manage the chosen team. This button would only show if the user was an administrator for that specific team.

To upload a file the user can choose a file with the button "Choose file". When the file has been chosen the button will show the file and the user can then click on "Upload file". When this is done the file is sent to the cloud and its meta data is sent to the database. To download a file a user clicks on a file in the file-section and then clicks on the "Download" button. If a user wants to delete a file then the user can choose a file and then click on the "Delete" button. The "Delete" button functionality has not not yet been implemented.

When a user clicks on a file in the file-section it turns blue, to make it easier for the user to understand which file has been selected. There is some information attached to each file, the file name, the owner of the file, the file size and the upload date of the file. Some of this file information is not currently shown in the main view.

When the user wants to log out they can use the "Log out" button. In the upper right corner there is a navigation bar where the user can choose between "Home" and "Profile". If the user clicks on "Home" the main page will appear and if the user instead clicks on "Profile" a new page is supposed to appear where the user could manage its application information, this has not been implemented.

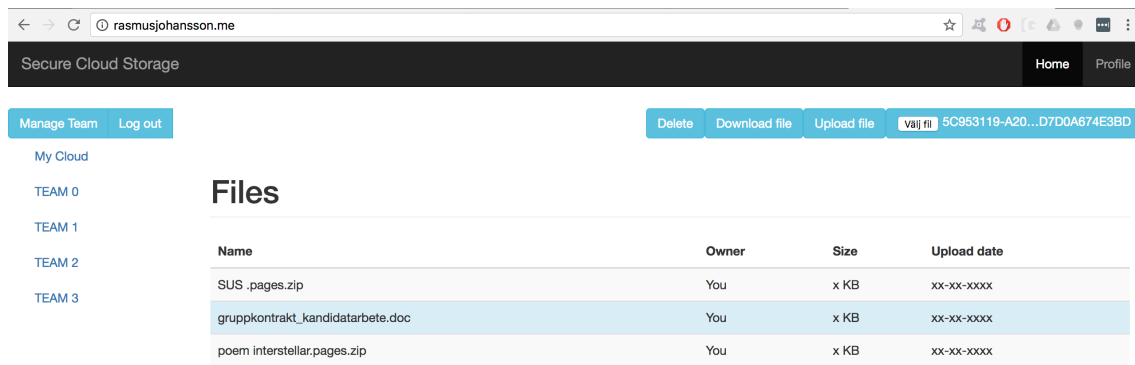


Figure 3: The main page of the application where one file is selected.

4.2 High-level architecture

The high-level architecture of the prototype can be viewed in Figure 4. It consists of a front-end which runs in the client's browser and a back-end server which communicates with the front-end. The back-end also consists of a database that keeps track of the users and which files belong to which user. The back-end sends and fetches the relevant files from the external cloud. All files will be stored in the same file-area in the cloud.

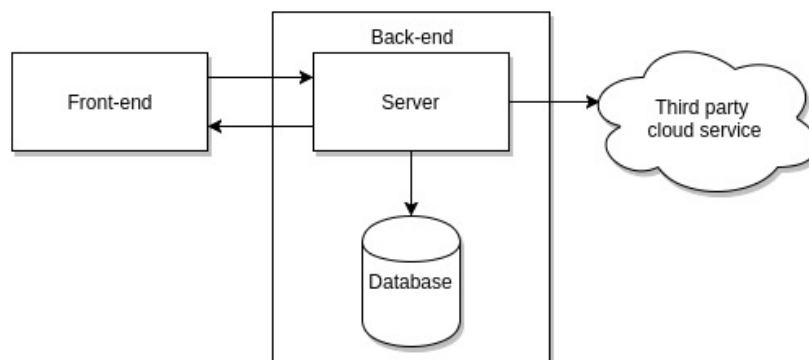


Figure 4: Diagram of the high-level architecture.

4.3 How security is achieved

This section will present the scheme that has been devised in order to provide security to the users of the product. It will also feature flow-charts of different procedures within

the system. All abbreviations used in the explanations of the cryptographic parts can be found in Table 1. It should be noted that this is the intended solution, everything is not implemented in the product as of yet.

The only security feature that is currently implemented is the encryption and decryption of files using AES. There is one flaw in the current implementation which is that for each file that is encrypted, a new key and initialisation vector are generated. However, only the most recently generated key and initialisation vector are stored, and they are stored in the application memory only.

A consequence of the current implementation is that if more than one file is uploaded during a session, only the most recently encrypted file can be decrypted. Also, the key and initialisation vector are not being saved between sessions. This results in the problem that files which were uploaded during earlier sessions are now unusable since the keys for these files are lost.

The current implementation is not secure and only works as proof of concept in the sense that it shows that the cryptographic library Forge has been imported and is working as intended. More about Forge in section 4.4.3.

<i>pw</i>	passphrase
<i>mk</i>	main key
<i>fk</i>	file key
<i>pk</i>	public key for asymmetric cipher
<i>sk</i>	secret key for asymmetric cipher
<i>AES</i>	symmetric encryption algorithm
<i>RSA</i>	asymmetric encryption algorithm
<i>SHA3</i>	hashing algorithm
<i>KDF</i>	key derivation function

Table 1: Dictionary for abbreviations used in the security explanation

The notation $AES(k, m)$ means that the message m is encrypted using the AES algorithm with the key k . The main key will be generated from the passphrase of the user when they sign in, $mk = KDF(pw + \text{"mainke"}, \text{salt})$. This key will be used to store any other keys associated with the users account on the server. The server also has its own RSA key pair. The following information will be stored on the server for each user:

- pk
- salt
- $KDF(pw + \text{"pass"}, \text{salt})$
- $AES(mk, sk)$

- $AES(mk, fk_i)$, for each file i

At the beginning of each session a session key, AES key, is generated and shared through RSA. AES is primarily used for the communication since AES is more efficient than RSA. The following is also appended to all messages for the sake of message integrity and sender verification:

- $SHA3(c) = h$, where c is the sent ciphertext.
- $RSA(sk_{sender}, h)$, where h is the hash of the ciphertext.

4.3.1 Registration

At registration, an asymmetric key pair (pk_{user}, sk_{user}) and a salt is generated. The following information exchange will take place between the client and the server as a result of registration. After registration a user is considered logged in.

1. Client \rightarrow Server: $RSA(pk_{server}, m)$
 $m = key_{session} + pk_{user}$
2. Client \rightarrow Server: $AES(key_{session}, m)$
 $m = \text{given name} + \text{family name} + \text{email} + \text{salt} + h$
 $h = KDF(pw + \text{"pass"}, \text{salt})$
3. Client \rightarrow Server: $AES(key_{session}, m)$
 $m = AES(mk, sk)$

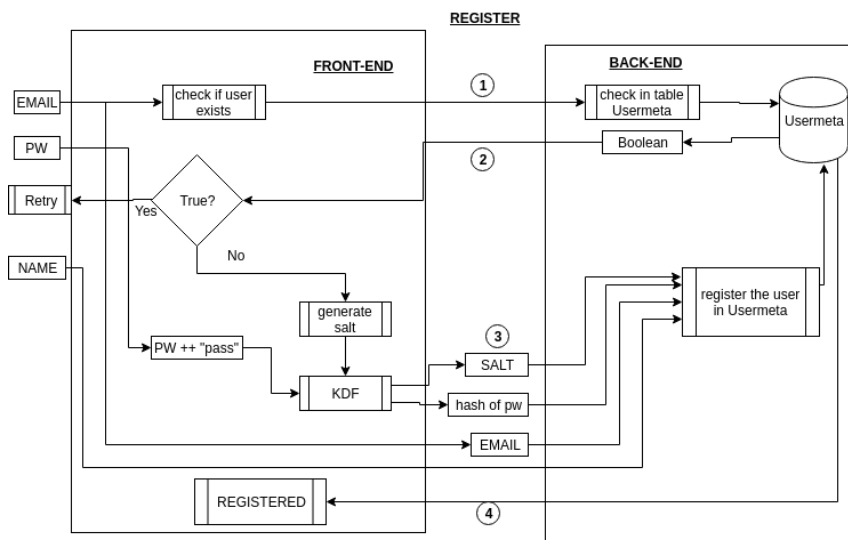


Figure 5: Flow-chart of Registration procedure

The procedure of registering a user shown in Figure 5 is completed in four steps.

1. The user inputs email, pw and name in the registration form. Upon clicking the register button, a request is sent to the back-end to check in the database that the email is not yet registered.
2. A boolean is returned from the back-end, if it is true the user is asked to retry (the email is already registered). Otherwise the pw is concatenated with the string "pass" and passed to the KDF alongside a generated salt. The reason for appending "pass" is to distinguish the string from the one used to generate the mk . The KDF generates a hash of the $pw + "pass"$, with the salt (a bit string) to increase strength against dictionary attacks. The output of the KDF is a hash and is used by the back-end to be compared to the hash generated from the user-inputted pw upon login.
3. The email, salt and hashed pw belonging to the user is sent to the user meta of the database and registered there.
4. A notification is sent to the user alerting them that the registration was completed, and the user is considered logged in.

4.3.2 Login

Login is performed by passing the email and hashed pw to the server after a session key has been established.

1. Client \rightarrow Server: $\text{RSA}(pk_{server}, m)$
 $m = key_{session} + pk_{user}$
2. Client \rightarrow Server: $\text{AES}(key_{session}, m)$
 $m = \text{email} + h$
 $h = \text{KDF}(pw + "pass", \text{salt})$

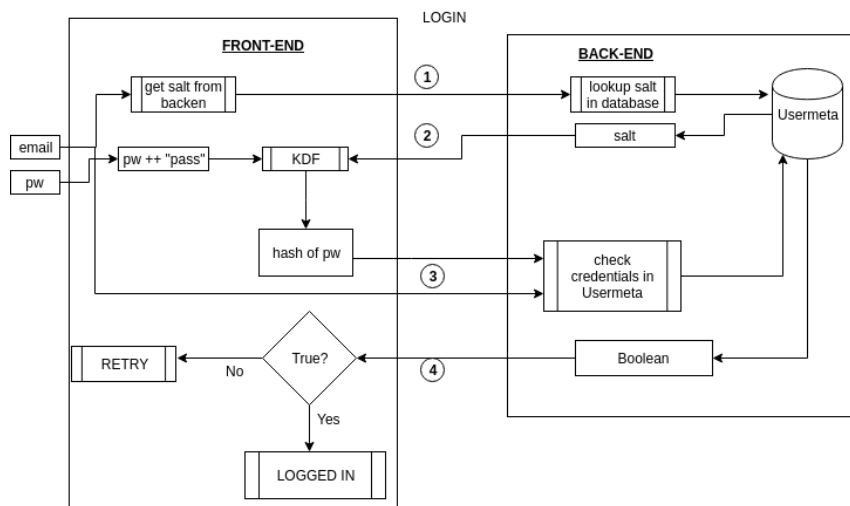


Figure 6: Flow-chart of Login procedure

As seen in Figure 6 the login procedure has four fundamental stages.

1. The user provides email and pw . When the login button is pressed, a request is sent to the back-end to look up the salt associated with that email.
2. The salt and the pw concatenated with the string "pass" is sent to the KDF which generates the hash of the pw .
3. The email and hashed pw are checked against the credentials saved in the user meta withing the database. The hash is used to isolate the back-end from the actual pw .
4. A boolean is returned from the database. If it is true the user is logged into the system. If it is false the login attempt fails and the user will need to retry.

4.3.3 Uploading a file

As a new file is selected for upload, the bytes of the file are read into an array buffer. This buffer is then encoded as an array of 8-bit unsigned integers. Since Elm only supports sending strings over WebSockets, we chose to encode the array as a UTF-8 string. The string is then passed on to be encrypted with a newly generated file key fk . The message $AES(key_{session}, m)$, where $m = AES(fk, file) + AES(mk, fk)$ is then passed to the server. It is assumed that a session key has already been established at login.

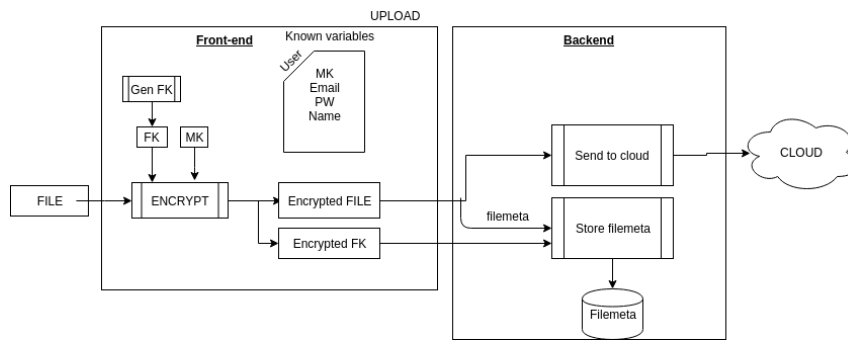


Figure 7: Flow-chart of Upload procedure

The procedure of uploading a file is visualised as a flow-chart in Figure 7. First, the user chooses a file to upload. When the user presses "upload", a unique fk is generated and used to encrypt the file. This way each file gets an unique key which, if shared, can't give access to anything other than that file. Then the user's mk is used to encrypt the fk . These two encrypted objects are then uploaded to the back-end. This way the server can store the encrypted fk , to which only the client has the key. The database knows who the user is since the start of the session.

4.3.4 Downloading a file

It is assumed that prior to downloading a file, a session key has been established at login.

1. Client \rightarrow Server: $\text{AES}(key_{session}, \text{filename})$
2. Server \rightarrow Client: $\text{AES}(key_{session}, m)$
 $m = \text{AES}(mk, fk) + \text{file}$

The downloaded file is then decrypted with the fk and converted back from an UTF-8 string into an array of 8-bit unsigned integers. The array is fed to a JavaScript blob (binary large object), which is a file representation in JavaScript, and written to a file which is placed in the default download folder of the browser. A flow-chart of this procedure is visualised in Figure 8.

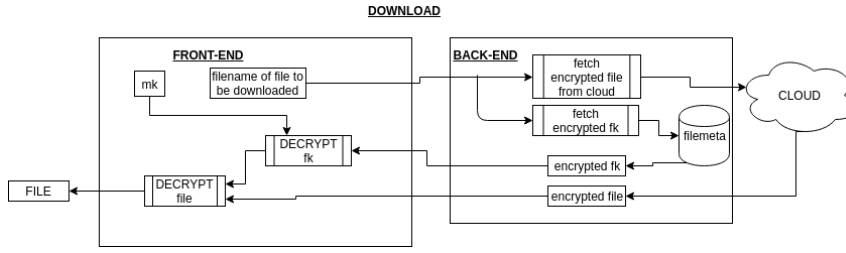


Figure 8: Flow-chart of Download procedure

4.3.5 Sharing a file

As a member is invited to a team, the following process will take place for each file that needs to be shared. The current member is referred to as Client1 and the newly invited user is Client2.

1. Server \rightarrow Client1: $AES(key_{session}, m)$
 $m = AES(mk_{user1}, fk)$
2. Client1 \rightarrow Server: $AES(key_{session}, m)$
 $m = user2 + RSA(pk_{user2}, fk)$

The next time user2 signs in, the following exchange will happen without any need for user interaction.

1. Server \rightarrow Client2: $AES(key_{session}, m)$
 $m = RSA(pk_{user2}, fk)$
2. Client2 \rightarrow Server: $AES(key_{session}, m)$
 $m = AES(mk_{user2}, fk)$

If a file is added to an existing team, the same procedure will take place for each user in the team.

In the case where a user is kicked from or leaves a team, all the files that are shared with that team need to be uploaded again in accordance with the procedure in Section 4.3.3 and with new file keys.

The sharing of a file takes some iterations between the client and the server, but can be explained with Figure 9. Breakdown of the figure:

1. The name of the file to be shared is sent to the back-end to retrieve the fk of the file. This is stored encrypted with the user's mk and needs to be decrypted from ciphertext to plaintext.
2. The sharer needs to know the pk of the users whom the file is intended to be shared with. Then the fk is duplicated for each user and asymmetrically encrypted with that user's pk .
3. The encrypted fks are sent to the back-end in order to be stored in the database for respective user.
4. Here User1 is used as an example. User1 retrieves the encrypted fk from the database and decrypts it with their sk .
5. Now User1 can encrypt it again with the mk and store it in the database. With this protocol the encrypted fks stored for each user on the server is only able to be decrypted into plaintext again using the specific user's sk .

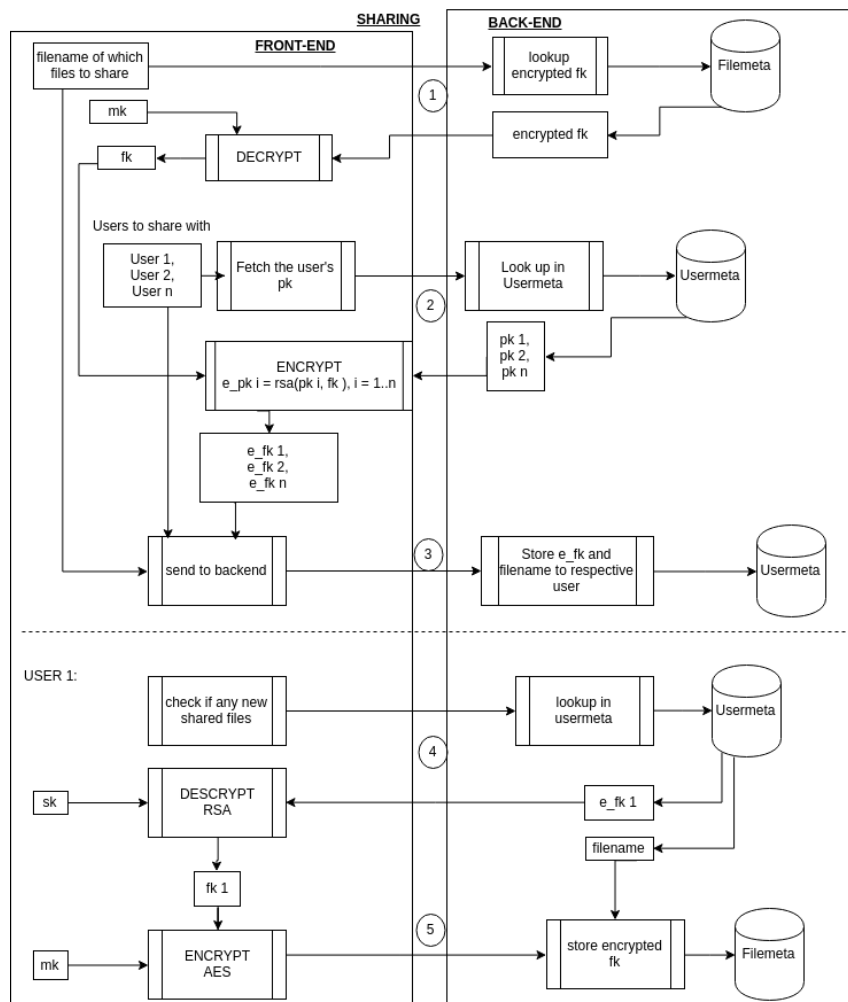


Figure 9: Flow-chart of Sharing procedure

4.3.6 Changing passphrase

The passphrase is used for generating the main key which in turn is used to store a user's keys on the server. The main key has no connection with the stored files. This means that the only action that is necessary when changing password is re-encrypting the users secret key and the file keys with the new main key. Since the amount of data is relatively small and it is encrypted with AES, this is not a problem. Though this functionality has not been implemented yet.

4.4 Front-end

Since the intent behind the product is to move the encryption client-side, most of the application logic ends up in the front-end. The front-end is implemented in a purely functional domain-specific language called Elm and the communication with the server goes through WebSockets. Following is a description of the design decisions and thoughts that lies behind the code structure. Furthermore, there will be an explanation of how the client-side encryption is implemented.

WebSockets is a protocol enabling a two-way communication between the client (front-end) and the server (back-end), the protocol is initialised with a handshake followed by basic messages layered over TCP [21]. This means that when a connection is established over a pre-defined port, messages can be sent freely from the front-end to the back-end and vice versa. We chose to use WebSockets for the client-server communication due to the difficulties with file manipulation in Elm. Encoding the encrypted files as strings and passing them over WebSockets simplified the uploading process significantly in comparison to previous attempts with HTTP.

4.4.1 How Elm was chosen

Initially the idea was to have a type safe solution by generating both the front-end and back-end using Haste.App. However, due to lack of progress and technical issues, discussed further in Section 5.9, a decision was made to replace Haste.App. To find a suitable front-end replacement, a number of requirements was established:

- Crypto library with various functionality, described briefly below.
 - Asymmetric encryption is necessary for passing keys for file sharing.
 - Symmetric encryption is needed to encrypt files before uploading them.
 - Key derivation function is important, otherwise the generation of main keys is impossible.
 - Hashing is needed to store passphrases securely in the database.

- There is also a need for authentication, to be sure the correct user is signed in.
- Pseudo random number generator, PSNG, is needed to generate the filekeys.
- A source of entropy is necessary to generate randomness through the PRNG.
- For obvious reasons it is important that the compiled code of a web application is runnable in a web browser.
- Graphical elements are needed to present files, but also to illustrate a clean and intuitive file structure.
- The front-end needs to be able to read user credentials, both upon sign up and login.
- There is also a need for local file browsing, to choose the file to upload.
- Reasonably simple handling and manipulation of graphical elements, enabling design of a user friendly interface.
- Communication with the back-end is imperative, for sending and receiving information upon activities like signing up and logging in.

Since the program is intended to run in a web browser environment JavaScript would have to be involved in some way and so two main alternatives was derived; writing the code directly in JavaScript or using something that compiles to JavaScript.

Ultimately Elm, which is a domain specific language that compiles to HTML, CSS and JavaScript, was chosen as the preferred front-end [22]. The choice was made with the requirements in mind. The cryptographic properties of the front-end is of great importance to provide the desired functionality of the product. They are in different ways involved in file sharing, signing up, logging in, encrypting files and so on. However, Elm in itself does not have much to offer regarding cryptographic packages but there is functionality to import JavaScript functions which makes it possible to make use of any JavaScript based crypto library. The requirement of a reliable source of entropy can also be met by importing JavaScript functions. Graphically presenting files, reading user credentials and browsing local files is also possible to achieve within Elm or through the imports of JavaScript functions, as well as handling graphical elements. Communicating with the back-end is possible within Elm. Hence, Elm meets the requirements of our front-end.

The decision was also partially based on personal preference. Eliom, a competitor to Elm, was quickly ruled out due to being built on OCaml, which none of the team members have any prior experience of [23] [24]. React in combination with Redux is a viable alternative to create the program completely in JavaScript [25] [26]. However, JavaScript in itself is not very developer friendly in a way that it lets the programmer get away with code containing erroneous behaviour. This in turn can cause runtime errors, which are costly. Elm on the other hand does not allow this type of errors due to being type safe, resulting in smooth running applications. Elm also offers better performance than React, which is a plus. This combined with the team's previous experience of typed functional languages influenced the choice of Elm.

4.4.2 The Elm architecture

Elm is currently in an early state, version 0.18, which means not all of the functionality that you expect from HTML, CSS and JavaScript is currently available, e.g. file manipulation. Elm has solved this problem through an abstraction called ports. Ports allows for interoperability with JavaScript, which means that anything that is missing in Elm can be written in JavaScript instead.

Elm has a built in architecture that splits every application into three parts: model, update and view. The model is a record which keeps the current state of the application, the update function updates the state of the application based on events, and the view listens to the model and determines what to show based on the current state of the application.

The events that are consumed by the update function can be fired from the view, e.g. a button press, or from subscriptions. Subscriptions is Elm's way of letting systems outside of Elm communicate with Elm, and in our case it is mainly used for receiving websocket messages from the server and receiving return values from the parts written in JavaScript.

The majority of the CSS used in the application comes from the framework Bootstrap [27]. It is downloaded to the user client through a content delivery network, CDN, which is shown in Figure 10. To use the style sheet which is using Bootstrap, you only need to include it in the view with the type Html Msg. In Figure 11 it is illustrated how the style sheet is used in the function view.

```
stylesheet : Html Msg
stylesheet =
  let
    tag =
      "link"
    attrs =
      [ attribute "rel" "stylesheet"
        , attribute "property" "stylesheet"
        , attribute "href" "https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
      ]
    children =
      []
  in
    node tag attrs children
```

Figure 10: How the Bootstrap CSS is imported.

```
view : Model -> Html Msg
view model =
  div []
    [
      --NAVIGATION
      navigationbar,
      --MAIN CONTENT
      div [class "container-fluid"]
        [fileNav model,
          div [class "row"]
            [
              sidebar, (center model)
            ]
          ],
      stylesheet
    ]
```

Figure 11: How the stylesheet is used in the view function.

4.4.3 Cryptography in the browser

As previously mentioned client-side security is a major part of this project. Since Elm does not have a lot to offer regarding cryptographic libraries this functionality had to be filled by importing a JavaScript library, Forge [28]. Forge is an extensive cryptographic library and it contains implementations of all the cryptographic primitives that was needed for this project.

Forge only supports one algorithm for public key encryption, RSA [29]. Only having one alternative is not a significant problem in this case, since RSA has been approved as a standard by NIST [30]. Furthermore, there is a good chance RSA is the world's most used public key cryptosystem. One reason for this is its versatility, as RSA can be used as a tool in both digital signatures and public key encryption [8].

Forge offers three different block ciphers in the forms of RC2, DES and AES. However only the latter is relevant to this project since DES has been proven inadequate and RC2 is considered vulnerable [8] [31].

A variety of hash functions are implemented in Forge. Including MD5 and SHA-1, which should not be used since both of them are considered broken [8]. However, the secure hash functions SHA-256, SHA-384 and SHA-512 are also included. The three of them all use the SHA-3 as underlying hashing algorithm [32] [33].

Forge includes an implementation of the password based key derivation function 2, PBKDF2 [34]. PBKDF2 is part of the Public Key Cryptography Standards, PKCS, series created by RSA Laboratories. PBKDF2 works by taking a password, a salt, an iteration count and desired key length as inputs, it then derives a key from these inputs by applying a pseudo random function. Optionally this pseudo random function can be passed as a fifth input to the key derivation function [35].

Forge provides a PRNG based on the Fortuna algorithm. It also comes with an API for collecting entropy, but if possible the internal browser function `window.crypto.getRandomValues` will be used instead [28]. The Fortuna algorithm works in three stages. First it creates random amounts of pseudorandom data from a fixed-size seed through its generator. Fortuna also has an accumulator that is capable of collecting and pooling entropy. The third stage is seed file control, this makes it possible to generate random data shortly after re-starting the computer [8].

Providing functionality within all the branches of cryptography needed for this product is a significant challenge though the vast capabilities of Forge made things easier since there was only one library needed for all the cryptography primitives. Forge has another advantage, according to benchmarking it is fast [36] [37]. Performance is not the primary concern when discussing crypto libraries, but still a relevant quality since it affects the usefulness of the library. The importance of reasonable performance also increases with the number of complex crypto operations performed. However, the benchmarking results from the speed test should be reviewed with some caution since the test itself has not been updated for several years and may be outdated, there is also little information about how it has been implemented.

4.4.4 Algorithm choices

Regarding cryptography algorithms there were several choices to be made. AES was chosen as preferred block cipher algorithm, used for symmetric encryption. The decision was based on AES being the official standard and partially due to AES having good performance [8]. On similar grounds RSA was chosen for asymmetric encryption purposes, it is the standard algorithm and also very widely used. For hashing SHA-3 was chosen. Though SHA-2 is a good alternative and an approved standard, SHA-3 is the latest standard [38]. Hence, SHA-3 is the more modern of the two and there is not really a reason to use SHA-2 over SHA-3 in a new and modern application.

4.4.5 Choice of block cipher mode of operation

Regarding block cipher modes of operation there had to be a decision of which one to go for. Mainly there were two alternatives, either CBC or CTR. Both these modes have been approved by NIST. They are recommended for use in cryptographic systems in combination with a symmetric key block cipher algorithm that has been approved by Federal Information Processing Standards, for instance AES[39].

For this project CBC, with a random IV, was chosen instead of CTR. The reason for this is that nonce generation is perilous and a serious source of security weaknesses, due to not being able to guarantee that the nonce is unique. This problem is not unique for CTR, it is also present when using CBC with nonce generated IV. However, it can be avoided by using a random IV instead. In general CBC is a good mode, it is robust and can resist some abuse [8].

However, CTR is also an interesting alternative. It has a clear advantage over CBC, performance. When using CBC the encryption process has to be done serially, since the encryption of each block is dependant on the one before. However, in CTR this is not the case and the encryption can be parallelized [39]. This can increase performance and in turn also usability of the application.

4.4.6 File manipulation

File manipulation and file I/O is something that is not yet implemented in Elm. At first an Elm-package file-reader, which is a ported version of JavaScript's file-reader, was used for file manipulation. Later this was changed to plainly use the actual JavaScript package.

4.5 Back-end

The back-end currently consists of mainly two different modules, the database and the websocket handler. The websocket handler is an object class which is instantiated for each connection request. As a request is received from the client-side it is split up into two parts, the request type and the required parameters. For example the request "*login|email|passphrase*" is split into *login* and a list of parameters [*email, passphrase*]. The handler then forwards the parameters to the responsible module. The server response will be sent asynchronously when the request has been processed, e.g. "*login|ok*".

There are other modules planned but not currently implemented. Firstly, there will be a module that handles all interaction between the server and the cloud storage. This module is currently in prototype stage. Secondly there will be a module that implements or utilises a library that contains the algorithms AES, SHA3 and RSA. These algorithms are needed server-side in order to encrypt the data sent over the web socket and verifying

the sender and the integrity of sent messages.

4.5.1 Python - Back-end language

Perl, PHP, Ruby, Node.JS, Java and Python was all researched and weighted against each other, but ultimately Python was chosen as back-end language. Python is a general-purpose language widely used for writing the server-side of various applications. This means that the necessary functionality is possible to implement and there exists plenty of resources that will ease the implementation.

With Python the back-end goals were achieved and are done in a modular way. With Python there should also not be any implementation restraints on what has not yet been implemented.

4.5.2 Tornado - Communication between front-end and back-end

The back-end was first done in Django Python, but Django had too much extra functionality that was not needed, so the back-end was made in Tornado Python instead.

Django works by having a folder, called "app", with a number of files in them for each major functionality and/or "path" of the website which was unnecessarily complex. Django also only works with html-files as standard which was troublesome.

Tornado is much simpler and lightweight as it only uses a python class for each handler and then some extra code for defining which port to listen to and which handler each path should use.

4.5.3 Database

All metadata is stored on the back-end in a MySQL database [40]. The databases contains two tables, "Usermeta" and "Filemeta". Usermeta is a simple table without any special features and just stores metadata regarding the user with columns for name, email, hashed password, salt, encrypted secret key and public key. However the columns for salt, encrypted secret key and public key are not yet implemented in the product. Filemeta is a table which stores metadata about the files, with the columns for name, size and owner. The owner column is a foreign key field which points to a certain row (user) in the Usermeta table. This way there is one table for all the users, Usermeta, and one for the files, Filemeta, with a mapping between a user and their files through the foreign key field "owner" for each file. A picture of the database and its tables are visualised in Figure 12.

DATABASE			
FILEMETA			
Filename	Size	Owner(represented index of user in Usermeta)	
file1	32	3 (Loket)	
file2	64	1 (Glenn)	
file3	...	2 (Kroner)	
file4	...	2 (Kroner)	
file5	...	4 (Strömberg)	
USERMETA			
Index	Name	Email	Pw
1	Glenn	glenn@...	aedf3ac212ab
2	Kroner	kroner@...	...
3	Loket
4	Strömberg

Figure 12: Representation of the database and its tables

The communication with MySQL and constructing of MySQL-queries was made with the python library Peewee, which is an api for executing queries in a more intuitive way using specific functions instead of constructing queries on your own [41].

4.6 Cloud Storage

One of the main selling points of the project is to connect the powerful infrastructure of the big and well known cloud service providers with the security of client-side encryption. To simplify the development process and reduce legal issues the group has for the sake of the prototype in this project instead chosen to use an open source cloud service solution. This means that the group has full control of the cloud service similarly to what it would be like if there was a collaboration with a larger cloud service company. There has also been put emphasis on making all development towards the cloud service modular so that a swap of cloud service would be simple.

The cloud is currently also developed with a dummy storage functionality, where the files are saved in the same repository as the code. The cloud service is up and running but has not yet been connected to the uploading functionality on the website.

An open source cloud, Owncloud, has been set up as a placeholder cloud service to receive files [42]. Owncloud has an API which works with python which the back-end is done in and the hosting of the cloud is done on the same server as the back-end and so the cloud compatibility is only a few lines of code which saves the file in an organised matter on the cloud. Owncloud has a structure of individual user accounts but these are not used in our implementation. We instead store all of our data in a single Owncloud-users account and keep track of who owns what file by the use of our database with metadata.

The choice of Owncloud was made by looking at some prerequisites. The cloud service should be open source and free to use. Furthermore that it was also possible to control it via some python-API to guarantee it being compatible with the rest of our back-end.

5 Discussion

In this section there are discussions about the key issues of the project and also some security discussions about the level of security in external libraries and how the systems security could be enhanced with forward secrecy. There is also evaluations of the languages which have been used in the project and an ethical discussion at the end.

5.1 Security considerations

As mentioned in Section 1.4.2 there was no intent to implement any cryptography primitives ourselves. This is since our current knowledge of cryptography is not sufficient enough to do so. The implementation of these primitives would probably have been considerably less trustworthy than an established cryptographic library. Hence, creating a cryptographic library of your own would risk compromising the security of the system.

There are several risks with using an external cryptographic library. Some of these risks are bugs within the library, incorrect implementation of cryptographic primitives and a misunderstanding of how the library is meant to be used.

Bugs are deemed unlikely since Forge has been around for almost four years, which is plenty of time to find and correct early stage bugs [43]. Secondly, Forge has no less than 17 contributors, which suggests that bugs caused by tunnel visioning are unlikely.

Judging whether the implementation of the cryptographic primitives are correct or not is very difficult. The reason for this is that it is very hard to do reliable testing in Cryptography [8]. It is possible to send a test vector and check the output to determine if the result is correct. However, this only shows that the algorithm at hand works as intended, it does not guarantee that the system is secure.

For the system to be secure the entire solution has to be implemented properly. Forge is well documented and this helps prevent misuse but the risk is always there. To minimise this risk the group has carefully researched how to make proper implementations with Forge.

Apart from the cryptography library there are other libraries in use. Tornado that handles the back-end part of the communication between the front-end and the back-end, and Peewee that makes it easier to make MySQL queries in Python. Tornado can not compromise any information as it only handles encrypted data. If someone would break the communication the result would only be a possible sabotage of information. Peewee follows the same principle of only allowing for an intruder to destroy information rather than accessing it. If someone would be able to change the database through Peewee they could remove the information of what users are linked to what files. This would make our system lose track of who owns what and thereby making the files non accessible.

A general thing to keep in mind is that the testing of security systems is very difficult. Due to this and the limited time frame of the project there have not been enough time to include any practical testing of the cryptographic solution. It has however been theoretically evaluated and deemed to not have any obvious security flaws. In conclusion the service is secure if the assumptions regarding Forge as well as the theoretical evaluation hold.

With our current thought out solution the access of a user's information is completely dependant on that user's passphrase. If the passphrase is compromised that user's account is compromised as well. This solution is used because then the user can reach their files from any machine as long as they has memorised the passphrase. This would not not have been possible if the cryptographic keys would have been kept locally on each machine instead.

5.2 Security vulnerabilities

We have identified some problems regarding the security of our application that we have not had the time to come up with solutions for. Firstly, an adversary could sabotage the stored information by replicating an old session. Consider the scenario where a user uploads a document and then at a later point in time uploads a new version of the same document, overwriting the first version. If an adversary re-sends the encrypted data from the first session, the document will be overwritten by a prior version and the newer version is lost.

Secondly, there is a problem with the sharing protocol that can result in compromised data. As a client requests another users public key, in order to share a file key with another user, an adversary could intervene. If the adversary pretends to be the server and supply their public key to the client, then the client is essentially handing out the file key to the adversary. This can be solved if the server works in trusted mode by having the server sign the sent data. In the case of a server working in un-trusted mode, we have not solved how to store a secret on the server that can be used for verification.

5.3 Forward secrecy

One problem with the currently planned encryption scheme is that if at any point an adversary acquires the means to decrypt the RSA encryption that is used at login, see section 4.3.2, they can read all messages sent to or from all users. This is since at login, the client sends a session key to the server by encrypting it with the public RSA key of the server. Since the server's public key is the same for all users, all the data that is stored with our application is compromised. Cracking the session key implies accessing the emails and hashed passwords which in turn implies full access to all data.

An optimal solution should imply that if a session is compromised, none of the previous session nor any future sessions should be compromised. This is commonly referred to

as forward secrecy and there are handshaking algorithms that provides this. In the case of further development, after the end of the project, a solution which provides forward secrecy should be implemented.

5.4 Scalability

Regarding the scalability of the application, primarily two aspects are being taken into consideration. Those two aspects are the amount of users of the application and the amount of data traffic passing through our system. Regarding the amount of data stored, it lies mostly with the restrictions from the third-party cloud provider.

In our current solution, all files that are uploaded to the cloud passes through our back-end. This is not a good solution since it scales very poorly with increasing amounts of data traffic. A better solution would be to have the client requests cloud access from the back-end and then directly sends the files to the cloud. We have not come up with how to implement this without giving the client a permission that would also allow the user to sabotage the storage. The problem lies in giving the client write permission when uploading a file.

With regards to the amount of users, there does not seems to be any major problem. The only implication of increasing amounts of users, apart from the data traffic, is the amount of meta data we need to store and we believe that this is not a problem.

Lastly, we have the amount of users per team. Sharing a file with a team results in doing an RSA encryption of the file key for each receiver. It is not clear whether it is realistic to consider the performance problem caused by sharing a file with too many users. However, it can not be used to bottleneck the back-end since the encryption is performed client-side.

5.5 Evaluation of front-end

The decision to use Elm as front-end for our product was in general a good call. Learning Elm is rather easy, especially if the programmer in question has some prior experience of functional programming. It is an advantage to be familiar with the data structures commonly found in such languages. The syntax is different, but still reminds you of Haskell. Apart from the fact that Elm does not really care if you are an experienced programmer or a beginner. The reason for this is the architecture, Elm pretty much forces the programmer to use it. At first it was a bit cumbersome, but the more you get used to Elm the more obvious the brilliance behind the architecture becomes. The code is automatically neat and structured, which makes development so much easier compared to structuring the code yourself.

One thing to keep in mind is that Elm is young, for this project version 0.18 was used. Due to this there is some desired functionality missing. For example the support for file manip-

ulation is absent. As mentioned previously this problem was solved by using JavaScript for those parts. However, as soon as the ports and subscriptions were implemented correctly things went smoothly and importing JavaScript libraries was not much hassle at all. Another example of missing functionality are the WebSockets. Presently they only support strings, which means that all files need to be encoded to strings before being sent between the front-end and back-end. On the other hand Elm's age may be taken as a good thing. Being this good and user friendly while still being a comparatively young language shows that Elm has the potential to become something great.

5.6 Evaluation of back-end

There were two primary reasons for choosing Python as a back-end language. Firstly, Python has a large and active developer community and is commonly used for similar software projects. This means that there are numerous sources on the Internet where you can find help and suggestions for common problems. Secondly, due to the time constraint on the project it was necessary to choose a language with a low learning curve. Python has a general reputation amongst developers to be an easy language to learn. It also has similarities to Java, in which we have prior experience.

Moving away from the web framework Django and instead using Tornado for client-server communication was a good choice. We had no use for all the additional features that Django provided and therefore made the implementation with Tornado less complex.

5.7 Design principle and architecture of the back-end

As mentioned earlier in the current solution all communication between the user and the cloud passes through our back-end, seen in Figure 4. A clear disadvantage with this is the risk of massive data transfer, which could cause a negative effect on the usability of the system. The reason for this derives from the principle of using end-to-end encryption where only the user has access to their files. The only way to be sure of this is to have a single account at the cloud service operated by us. The user's files are then placed within the storage space of the central account of the cloud service. This is what makes the back-end a communication bridge between the cloud and the user.

It would be possible to let the user communicate with the cloud service directly. However, then there is no guarantee that a user can not access another user's files. Even if they are encrypted damage can be done in the sense of deleting sensitive information. This would not be acceptable, thus a compromise was made. Since security is the top priority of this project the decision to have the back-end work as a middle hand was clear.

5.8 Related work

There are several companies that provide cloud storage services with client-side encryption. Most of them however does not combine it with the infrastructure of bigger cloud storage services[4][5]. Our main competitor is called Tresorit and is the only competitor hosting a service that shares our vision of combining the two[6]. Their solution includes a native application and a web application, both which include client side encryption and the powerful third-party storage backend Azure hosted by Microsoft[44]. Tresorit also has sharing implemented in something that they call tresors. Tresors are very much like our thought out solution for including sharing with what we call teams. Just like our teams the tresors works as a file area where you can store files and then invite the people you want to have access[45]. Tresorit has a well developed solution for the same problem that we are trying to solve and due to being a leading company in the industry as well as having workforce of 50 employees it is very likely that their product will keep on improving[46].

5.9 Haste.App

Haste.App is a Haskell framework for creating web applications. It compiles Haskell code into both a JavaScript front-end and a binary back-end [47]. Haste.App was initially intended to be a big part of the project but got removed from the development process about two months into the project. The reason for its intended inclusion and later removal is discussed below.

5.9.1 Choosing Haste.App

Haste.App was initially introduced to the project group by the supervisors. It was pitched by the supervisors as an alternative to the more common way of writing a separate front-end and back-end. With Haste.App it is instead possible to write both front-end and back-end in the same file, all this in Haskell which has the advantage of including Haskell's type safety.

With this introduction and the fact that most of our group had more experience with Haskell than any commonly used language for developing web applications the group thought that Haste.App was worth a try.

5.9.2 Moving away from Haste.App

Haste.App started off slow with our group having quite a few different complications with the installation and initial setup. Installing Haste.App to use for compiling front-end was not a problem but when it came to also compiling the back-end some trouble occurred.

These setup problems occurred foremost because of the lack of documentation. After some existential troubleshooting it started working and development of prototypes was started. Though at this point the version of Haste.App that were being used, 0.5.5.1 (the current stable release), did not have support for any cryptography modules which was undoubtedly needed for the project. Anton Ekblad was contacted, the creator of Haste.App, and asked about this upon he provided a newer unreleased version. This version, 0.6.0.0, had support for a cryptography module and would in theory work with what was intended to develop.

Version 0.6.0.0 was however even more of a hassle to install and after trying to install it for over a week the decision that Haste.App was not the most optimal library for the project was made and new solutions was going to be researched. The primary problem with this installation was again the lack of documentation, for example there was no information about the need of external libraries.

The single biggest reason for having such big problems working with Haste.App is the fact that is a quite unknown framework. Few persons use it and thereby there is few posts about it on the Internet. The easiest way to overcome a problem in programming is to search for information about it on the Internet and this was simply not possible. The one place where you could get information about Haste.App was on the official Haste.App API and even there the documentation was limited.

5.10 Ethical perspectives

End-to-end encryption is a controversial topic and should not be taken lightly. Services where the data is fully hidden could be used for illegal activities such as trading illegal data and wares. Illegal data such as child pornography could be stored with a service as this one without anyone knowing. It is therefor important for the providing company to think about how to prevent these types of criminal activities. The alternative, to not use end-to-end encryption, has its own negative usage areas. Large cloud providing companies could monitor your data and even give or sell it to other parties such as governments.

End-to-end encryption could be an attractive property for some companies. One reason for this is to protect corporate secrets, both legal and illegal. A completely secure service of this kind could offer a significant step in prevention of industrial espionage. Consider a tech company for example; Imagine if the company has come up with some revolutionising new technology and wants to do everything they can to protect it against competitors. Then the end-to-end encryption property makes sure that the information is only accessible by the company's employees. In combination with a non-compete clause this would technically ensure that the information stays within the company.

6 Conclusion

Due to the unfortunate start caused by the problems with Haste.App, the prototype produced during the project is far from where we had hoped it to be. It does not provide a practically applicable version of any of the desired functionality that was planned. However, despite the lacking prototype implementation we have provided a foundation that should make further implementation relatively trivial.

The cryptographic library, the server database and the client-server communication are all working as intended. With regards to the modularity of the cloud, it should be rather easily exchangeable. Since Python is a widely used language for writing back-ends for web applications, most of the larger cloud storage providers have provided Python APIs for their respective service. For example the Google Drive API [48], the Dropbox API [49], and the Amazon S3 API [50].

We have planned and theoretically evaluated both our security protocol as well as the performance of our web application and believe that if the issues that are brought up in the discussion are addressed, we would have a working product.

References

- [1] T. Gaffer. (2016). Why client-side encryption is the next best idea in cloud-based data security, [Online]. Available: http://www.infosectoday.com/Articles/Client-Side_Encryption.htm#.WRV_3-XyiUk (visited on 2017-05-12).
- [2] Google. (2017). About google drive, [Online]. Available: <https://www.google.se/drive/about.html> (visited on 2017-02-09).
- [3] Dropbox. (2017). About dropbox, [Online]. Available: <https://www.dropbox.com/security> (visited on 2017-02-09).
- [4] Sync. (2017). Features of sync, [Online]. Available: <https://www.sync.com/features/> (visited on 2017-02-09).
- [5] Crypho. (2017). Main page, [Online]. Available: <https://www.crypho.com/> (visited on 2017-05-30).
- [6] Tresorit. (2017). Security of tresorit, [Online]. Available: <https://tresorit.com/security> (visited on 2017-02-09).
- [7] Therry Chia. (2012). Confidentiality, integrity, availability: The three components of the cia triad, [Online]. Available: <http://security.blogoverflow.com/2012/08/confidentiality-integrity-availability-the-three-components-of-the-cia-triad/> (visited on 2017-05-12).
- [8] F. Ferguson, B. Schneier, and T. Kohno, *Cryptography engineering*. Wiley Publishing, Inc., 2010, pp. 13, 43, 53, 54, 59, 63, 65, 66, 71, 72, 78, 79, 81, 82, 142, 195.
- [9] R. A. Mollin, *An Introduction to Cryptography, Second Edition*. Chapman and Hall, 2006, pp. 86, 91.
- [10] Python Cryptographic Authority. (2017). Asymmetric algorithms, [Online]. Available: <https://cryptography.io/en/latest/hazmat/primitives/asymmetric/> (visited on 2017-05-11).
- [11] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 1990, p. 165, ISBN: 9780138690175.
- [12] Mark Ward. (2015). Web's random numbers are too weak, researchers warn, [Online]. Available: <http://www.bbc.com/news/technology-33839925> (visited on 2017-05-12).
- [13] A. Gholipour and S. Mirzakupchaki. (2011). A pseudorandom number generator with keccak hash function, [Online]. Available: <http://www.ijcee.org/papers/439-JE503.pdf> (visited on 2017-05-12).
- [14] B. Kaliski. (2000). Pkcs #5: Password-based cryptography specification, [Online]. Available: <https://www.ietf.org/rfc/rfc2898.txt> (visited on 2017-05-12).
- [15] S. Alexander. (2012). Passwords matter, [Online]. Available: <http://bugcharmer.blogspot.se/2012/06/passwords-matter.html> (visited on 2017-05-12).
- [16] Scrum Alliance. (2016). Learn about scrum, [Online]. Available: <https://www.scrumalliance.org/why-scrum> (visited on 2017-02-09).

REFERENCES

- [17] Trello. (2017). Trello, [Online]. Available: <https://trello.com/> (visited on 2017-02-10).
- [18] Slack. (2017). Slack, [Online]. Available: <https://slack.com/> (visited on 2017-05-29).
- [19] Git. (2017). Main page, [Online]. Available: <https://git-scm.com/> (visited on 2017-05-09).
- [20] Git Hub. (2016). Hello world, [Online]. Available: <https://guides.github.com/activities/hello-world/> (visited on 2017-02-09).
- [21] I. Fette and A. Melnikov. (2011). The websocket protocol, [Online]. Available: <https://tools.ietf.org/html/rfc6455> (visited on 2017-05-03).
- [22] E. Czaplicki. (2017). Elm, a delightful language for reliable webapps, [Online]. Available: <http://elm-lang.org/> (visited on 2017-05-04).
- [23] Eliom. (2016). Eliom, [Online]. Available: <http://ocsigen.org/eliom/> (visited on 2017-05-12).
- [24] OCaml. (2016). Ocaml, [Online]. Available: <https://ocaml.org/> (visited on 2017-05-12).
- [25] React. (2017). React, [Online]. Available: <https://facebook.github.io/react/> (visited on 2017-05-12).
- [26] Redux. (2016). Redux, [Online]. Available: <http://redux.js.org/> (visited on 2017-05-12).
- [27] Bootstrap. (2017). Bootstrap, [Online]. Available: <http://getbootstrap.com/> (visited on 2017-05-09).
- [28] D. Bazaar/Forge. (2017). Forge, [Online]. Available: <https://github.com/digitalbazaar/forge/blob/master/README.md> (visited on 2017-05-08).
- [29] D. Longley. (2017). Rsa, [Online]. Available: <https://github.com/digitalbazaar/forge/blob/master/lib/rsa.js> (visited on 2017-05-08).
- [30] N. I. of Standards and Technology. (2013). Digital signature standard, [Online]. Available: <http://dx.doi.org/10.6028/NIST.FIPS.186-4> (visited on 2017-05-08).
- [31] M. Mathur and A. Kesarwani. (2013). Comparison between des , 3des , rc2 , rc6 , blowfish and aes, [Online]. Available: <http://www.met.edu/Institutes/ICS/NCNHIT/papers/33.pdf>.
- [32] D. Longley. (2017). Forge, [Online]. Available: <https://github.com/digitalbazaar/forge/blob/master/lib/sha512.js> (visited on 2017-05-08).
- [33] N. I. of Standards and Technology. (2015). Secure hash standard, [Online]. Available: <http://dx.doi.org/10.6028/NIST.FIPS.180-4> (visited on 2017-05-08).
- [34] D. Longley. (2017). Ket derivation function 2, [Online]. Available: <https://github.com/digitalbazaar/forge/blob/master/lib/pbkdf2.js> (visited on 2017-05-09).
- [35] B. Kaliski. (2000). Rfc 2898, [Online]. Available: <https://www.ietf.org/rfc/rfc2898.txt> (visited on 2017-05-09).
- [36] M. Rosica. (2012). Javascript cryptography speedtest, [Online]. Available: http://cryptojs.altervista.org/test/simulate-threading-speed_test.html (visited on 2017-05-10).

REFERENCES

- [37] D. Tarr. (2014). Performance of hashing in javascript crypto libraries, [Online]. Available: <http://dominictarr.github.io/crypto-bench/> (visited on 2017-05-10).
- [38] NIST. (2017). Nist releases sha-3 cryptographic hash standard, [Online]. Available: <https://www.nist.gov/news-events/news/2015/08/nist-releases-sha-3-cryptographic-hash-standard> (visited on 2017-05-11).
- [39] M. Dworkin. (2001). Recommendations for block cipher modes of operation, [Online]. Available: <http://dx.doi.org/10.6028/NIST.SP.800-38A> (visited on 2017-05-30).
- [40] Oracle. (2017). About mysql, [Online]. Available: <https://www.mysql.com/about/> (visited on 2017-05-12).
- [41] C. Leifer. (2016). Peewee, [Online]. Available: <http://docs.peewee-orm.com/en/latest/> (visited on 2017-05-12).
- [42] ownCloud. (2017). Owncloud, [Online]. Available: <https://owncloud.org/> (visited on 2017-05-08).
- [43] D. Bazaar/Forge. (2013). First release of forge, [Online]. Available: <https://github.com/digitalbazaar/forge/releases?after=0.1.6> (visited on 2017-05-12).
- [44] Tresorit. (2017). Third party services, [Online]. Available: <https://support.tresorit.com/hc/en-us/articles/216114397-Third-party-services> (visited on 2017-05-30).
- [45] —, (2017). Work securely within teams, [Online]. Available: <https://tresorit.com/business> (visited on 2017-05-30).
- [46] —, (2017). The tresorit story, [Online]. Available: <https://tresorit.com/about-us> (visited on 2017-05-30).
- [47] A. Ekblad. (2016). What is haste? [Online]. Available: <http://www.haste-lang.org/> (visited on 2017-02-09).
- [48] G. Developers. (2017). Python quickstart, [Online]. Available: <https://developers.google.com/drive/v3/web/quickstart/python> (visited on 2017-05-29).
- [49] D. developers. (2017). Dropbox for python, [Online]. Available: <https://www.dropbox.com/developers/documentation/python> (visited on 2017-05-29).
- [50] A. W. Services. (2017). Aws sdk for python (boto3), [Online]. Available: <https://aws.amazon.com/sdk-for-python/> (visited on 2017-05-29).