Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

Enabling Secure E-Mail Communication in Multi-Device Environments

Maximilian Meier

Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

Enabling Secure E-Mail Communication in Multi-Device Environments

Sichere E-Mail Kommunikation und deren Realisierung in Nutzerumgebungen mit mehreren Endgeräten

Author	Maximilian Meier
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Dr. Matthias Wachs
Date	February 15, 2016

Informatik VIII
Chair for Network Architectures and Services

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, February 15, 2016

Signature

Abstract

Email is a very important part in today's communication and users are used to access their mails anywhere, anytime, in particular with multiple devices. With this requirements, securing email communication with end-to-end-security is challenging since OpenPGP and S/MIME are designed for a use on one device only. To use these securing mechanisms on multiple devices, all necessary private, public, and trust information have to be provided on each system. Therefore a solution for a synchronisation must be found in order to enable secure communication in multi-device environments.

This thesis analyses and answers the questions about requirements, design, and implementation for a secure and extensible synchronisation mechanism to enable secure communication on multiple devices. To fulfil that goal, background information is presented and the core of the problem is analysed. In addition to that, the requirements are specified and an architecture how such an application can look is set up. Usable frameworks to reach the design are shown and a potential implementation for the application is presented. The outcome is a prototype that answers the research question and enables the synchronisation. In the end of the thesis the outcome is evaluated and upcoming problems are explained. The conclusion contains future work and how the application can fit in the modern communication.

Zusammenfassung

E-Mail ist ein wichtiger Teil der heutigen Kommunikation und Benutzer sind es gewohnt immer, überall und mit mehreren Geräte auf ihre E-Mails zuzugreifen zu können. Diese Anforderungen können Ende-zu-Ende-Sicherungsverfahren für E-Mails, wie OpenPGP oder S/MIME, nicht ermöglichen, da diese Verfahren für eine Verwendung auf nur einem Gerät gedacht sind. Um diese Verfahren zu verwenden, werden private, öffentliche und vertrauenswürdige Informationen auf den zu verwendeten Geräten benötigt. Daher muss eine Verfahren der Synchronisation gefunden werden, um sichere E-Mail-Kommunikation auf mehreren Geräten zu ermöglichen.

Diese Arbeit erstellt die Anforderungen, ein Design und eine Implementierung für einen sicheren und erweiterbaren Synchronisationsmechanismus, welche sichere E-Mail-Kommunikation auf mehreren Geräten ermöglicht. Um dieses Ziel zu erreichen werden notwendige Hintergrundinformationen präsentiert und der Grund für das Problem analysiert. Zudem werden alle Anforderungen erfasst und in einer Architektur verarbeitet. Außerdem werden einige Frameworks angeführt und es wird gezeigt wie diese implementiert werden können, um diese Anwendung zu realisieren. Das Resultat ist ein Prototyp, welcher die wissenschaftliche Frage beantwortet und eine Synchronisation ermöglicht. Zum Ende hin wird das Resultat ausgewertet und aufgetretene Probleme werden anschaulich erläutert. Die Schlussfolgerung beinhaltet, wie sich eine solche Applikation in die moderne Kommunikation einfügen kann und zukünftige Forschungsarbeiten werden erläutert.

Contents

1 Introduction 1
1.1 Goal of the Thesis 2
1.2 Outline 2

2 Background 5
2.1 OpenPGP 5
2.1.1 Implementation 6
2.1.2 Conclusion 7
2.2 S/MIME Specification 7
2.2.1 Key Elements 8
2.2.2 Implementations 9
2.2.3 Conclusion 9

3 Problem Analysis 11
3.1 Appearance 11
3.1.1 Multiple Devices 11
3.1.2 Mobile Devices 11
3.1.3 Sending Mails 12
3.1.4 Reading Mails 12
3.1.5 Trust Information 12
3.2 Approaches 12
3.3 Conclusion 13

4 Related Work 15
4.1 State of the Art 15
4.1.1 Synchronisation of OpenPGP Keys 15
4.1.2 Secure Communication in Closed Systems 16
4.2 Security Tools 16
4.2.1 KeePass 16
4.3 Summary 17

5 Requirements 19

5.1 Stakeholder Analysis 19
5.1.1 End User as Stakeholder 19
5.1.2 Organisation as Stakeholder 20
5.2 Use Case Analysis 20
5.2.1 Use Cases for an End User 20
5.2.2 Use Cases for an Administrator 21
5.3 Functional Requirements 21
5.3.1 Synchronisation 21
5.3.2 Consistency 21
5.3.3 Automation 22
5.3.4 Encryption 22
5.3.5 Configuration 22
5.3.6 Backend 22
5.4 Technical Requirements 22
5.4.1 Frontend 22
5.4.2 Backend 23
5.5 Design Requirements 23
5.6 Summary 23

6 Design 25
6.1 System 25
6.1.1 Description 25
6.1.2 Functionality 26
6.2 Backend 26
6.2.1 Server 26
6.2.2 Cloud Storage and Peer-to-Peer 27
6.3 Client side 28
6.3.1 Core Module 28
6.3.2 Plugin System 29
6.3.3 Client Side Communication Module 31
6.3.4 Synchronisation Side Communication Modules 33

7 Development Life Cycle 35
7.1 Native Implementations 35
7.2 Separation between Mobile and Desktop 36
7.3 Hybrid Implementation 36
7.4 Conclusion 37
7.5 Creation of a Development Life Circle 37
7.5.1 Suggested Frameworks 38
7.5.2 Conclusion 39

8 Implementation 41

8.1 Backend 41
8.2 Client 42
8.2.1 Desktop Environment 42
8.2.2 Mobile Environment 42
8.2.3 Application Environment 43

9 Evaluation 45
9.1 Multi Platform Support 45
9.2 Low Maintenance 46
9.3 Simplicity 46
9.4 Extensibility 46
9.5 Conclusion 46

10 Gained Experiences 49
10.1 EmberJS and PhoneGap 49
10.2 EmberJS and Electron 49
10.3 Electron and PhoneGap 50

11 Conclusion 51
11.1 Future Work 51

Bibliography 53

List of Figures

2.1 Encryption mechanism of OpenPGP 5
2.2 Verification mechanism of OpenPGP 6
2.3 Encryption mechanism of S/MIME 8
2.4 Verification mechanism of S/MIME 8

6.1 Diagram of the system 25
6.2 Server structure 27
6.3 Diagram of the application 28
6.4 Integration of core module 29

7.1 Diagram of development life cycle 40

List of Tables

7.1 Comparison of implementations 37

Chapter 1

Introduction

For today's communication, email is a very important element. Emails appear everywhere, while receiving newsletters or information about work or studies. But the technology is over 40 years old and security was not an intended goal of emails. Security mechanisms like transport security, using TLS [1], and end-to-end-security, using S/MIME [2] or OpenPGP [3], were defined much later. But these mechanisms are also over 20 years old. The way how people communicate changed a lot since then. 20 years ago, most people used mainly one device to communicate. OpenPGP and S/MIME fulfilled the goal of securing the email communication, because all necessary information, like encryption keys where available on the device that was used. An exchange of these information was not required because just one device was available for communication. The Internet changed dramatically since then. People today have several different devices, like smartphone, laptops, tablets, or desktop PCs, that are used to communicate. In todays communication, emails can be received anywhere and anytime on multiple devices. Users can also write emails whenever they want. This is possible because received emails are mostly unsecured and the availability on multiple devices is no problem. For that kind of communication OpenPGP and S/MIME do not match. Because to access an email that is secured by one of these techniques is dependent on essential information, like encryption keys, that are required to decrypt or encrypt an email. But these information are mainly stored on just one local device and are not synchronised across the other devices like the emails are. In order to reach a high adoption of OpenPGP and S/MIME, they have to fit in the modern way of communication and must be available on all devices at anytime. An additional problem is that the keys need a lot of effort to have them available several devices. But non-technical users should not be required to keep track of changes or even exchange the keys them self between devices. As a result of this, simplicity is very important for any change in today's communication. Otherwise a high adoption through out the users is not reachable.

1.1 Goal of the Thesis

The goal of this thesis is to establish a design and an implementation for a synchronisation mechanism that enables secure communication across multiple devices. Such an approach as two big requirements. These are simplicity of the application and availability on many platforms. In addition to that, the application has to run without user interference, with that everybody can use it easily and on all of his devices. To achieve this goal the following research question must be answered: What are the requirements and a suitable design to implement a secure and extensible mechanism, that allows secure email communication, using established End-to-End securing mechanisms like OpenPGP and S/MIME, on multiple devices? To cover this issue the following questions have to be answered before:

- Why do information have to be synchronised?
- What information have to be synchronised?
- What are the requirements for such an application?
- What application design can fulfil these requirements?
- How is a suitable multi-platform implementation achievable?
- How can the code be re-usable between different platforms?

1.2 Outline

The thesis is structured as follows: To understand what secure communication is about this thesis starts in Chapter 2 with background information on OpenPGP and S/MIME. The encryption procedures for both mechanisms are explained and the important parts are highlighted that need to be available on all devices. Also the current implementations are highlighted so that the approach of this thesis can adapt them. In Chapter 3 the problems are analysed why it is difficult to use OpenPGP or S/MIME on multiple devices. In Chapter 4, the state of the art is presented and how current approaches try to solve the problem of key management. Furthermore, tools are introduced which are used to secure important information. In Chapter 5 the technical, functional, and design requirements are defined. In Chapter 6, a suited approach for a design is introduced. With Chapter 7, a suitable implementation is researched and described. With this, knowledge, Chapter 8 highlights the important parts of the implementation of the thesis project. In Chapter 9, this implementation is evaluated how it suits the requirements and the design. In Chapter 10, issues that occurred during the implementation are described. These issues will help for other approaches in a multi-device environment. The conclusion contains information about the overall state of such an application and a possible topics for the future.

Chapter 2

Background

This Chapter gives background information, about the essential parts of securing mechanisms like OpenPGP or S/MIME for multi-device environments. With email security as an important topic, these two technologies enable end-to-end-security for mail. With this end-to-end security the mail is protected, that just the communication partners can read the information. The security is created with authenticity, integrity, and confidentiality of the information. But both are over 20 years old and are not adapted to a multi-device environment.

2.1 OpenPGP

OpenPGP [3] is a specification for securing email communication. The encryption and decryption of OpenPGP combines asymmetric, symmetric, and hash algorithms in a hybrid procedure. In the asymmetric part of the encryption, OpenPGP is using a pair of public and private key to secure information. So each attendant needs both, a private and a public key. The following figure shows the hybrid encryption procedure for OpenPGP. To encrypt an email the public key of the recipient is used. In addition to

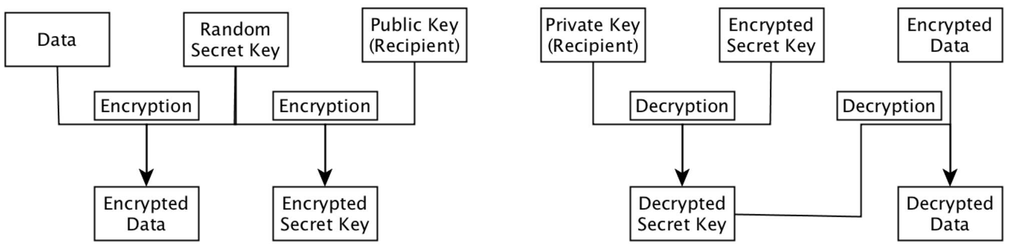

Figure 2.1: Encryption mechanism of OpenPGP

that a signature can be added to an encrypted email. This signature is created with the private key of the sender. A received email can be decrypted with the private key of the recipient. In order to check the senders integrity, the recipient can use the public key of the sender to verify him with his signature.

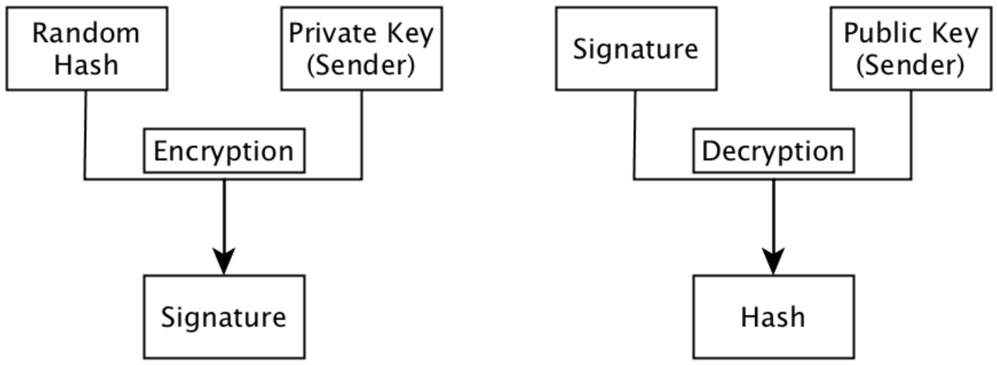

Figure 2.2: Verification mechanism of OpenPGP

2.1.1 Implementation

GPG (=Gnu Privacy Guard)[1] is an implementation of the specification RFC 4880 [3] of OpenPGP. This implementation is available through out the operating systems MacOS, Windows, and Linux and has different versions which are currently used. Version 1 and version 2 differ in the way how they are implemented. In GPG V1 everything is implemented inside and no external library is used. In GPG V2 the implementation uses external libraries. This implementations are often combined with tools like GPGTools for MacOS or GPG4Win for Windows. With GPG all information related to OpenPGP are saved in three files, a file for the private keys of a user, a file for the public keys of a user and his communication partners, and a file with the trust database, including all verifying signatures. It must be noted that in GPG version 2.1 the amount of files is reduced to 2 files, because the private key was moved to the file of the public keys. The file for the private key does still exist for backward compability, but new added keys are not inserted to private key file anymore. [4]

2.1.1.1 Key Elements

The key elements of GPG are the private key of the user and public keys of the user and his communication partners. Till version 2.1 these information are stored in separate files. One file contains all private keys of a user. Every new private key or sub key of a private key is added to that file. This file is named "secring.gpg". The public keys

[1]https://www.gnupg.org/

of a user and his communication partners are saved in a single file. This file is called "pubring.gpg". Every new generated public key of a user or the sub keys of a public key are added here. Also the public keys of new communication partners are stored here. For GPG version 2.1 the file "secring.gpg" is not used anymore. The information is now also stored in the file "pupring.gpg". [4] Even though the file "secring.gpg" still exists.

2.1.1.2 Trust Database

GPG is based on a web of trust. This means that authenticity of public keys is provided by the users of the Internet. Every user can sign public keys and is able to verify the owner of that key. All of this signatures are saved in a file on the local storage of the device. This file is called "trustdb.gpg".

2.1.1.3 Key servers

Key servers are the source of public keys of a user. If a user doesn't provide his public key directly, this key can be found with the help of a key server. Those key servers are not centralised by one authority. These servers are provided by several different organisations, like universities all around the world. It is important to note that these servers are just storages for public keys that can be accessed by the public. These servers do not fulfil the need of synchronisation of GPG information between several devices.

2.1.2 Conclusion

With the implementation of GPG, all information of OpenPGP to enable it in a multi-device environment are stored in two or three separate files on the local storage of the device, "secring.gpg", "pubring.gpg", and "trustdb.gpg". This is depending on the used version of GPG. To use the same data on a different device, these files need to be synchronised in order to enable OpenPGP on an additional device. This files are required for a GPG use on a mobile and desktop device.

2.2 S/MIME Specification

S/MIME [2] is a different securing mechanism for emails which is specified in RFC 5751 [2]. It is based on MIME types [2] which are also used by other information, like pictures or Base64 strings. The encryption is similar to OpenPGP because it also depends on a hybrid encryption and each communication partner also needs a public and a private key. Similar to OpenPGP the key pair can be generated by each person them self or with the help of a central authority. The difference is that central authorities provide the integrity of the public keys, by creating certificates for each communication partner. These certificates include information about the owner and the public key of the owner. As a result the authenticity is not generated with a web of trust but is provided by the central authorities. These authorities can be public or internal of a certain organisation. The way how encryption works in S/MIME is similar to OpenPGP

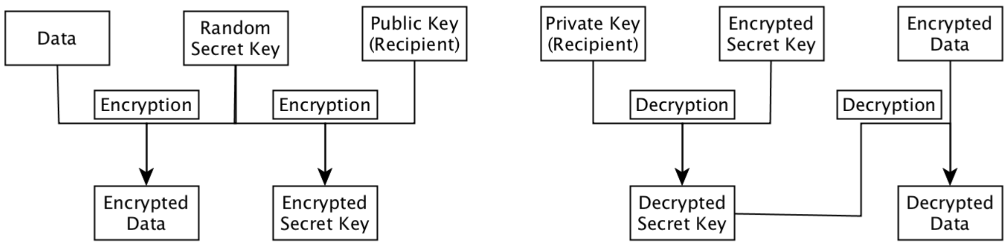

Figure 2.3: Encryption mechanism of S/MIME

because all communication partners have a private and a public key. The public key of the recipient is again used to encrypt the information of an email. With the private key of the sender a signature can attached to that encrypted email. The recipient can access the email with his own private key and can verify the sender with the public key of the sender. This public key is provided by the certificates which are distributed by the users them self or LDAP distribution services.

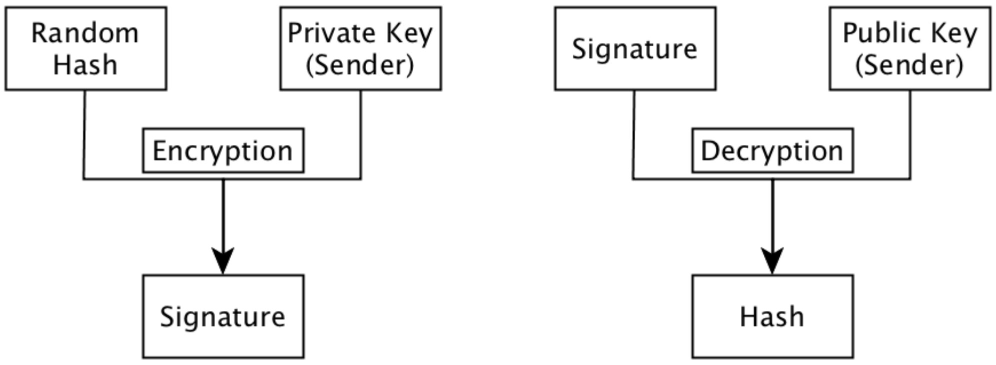

Figure 2.4: Verification mechanism of S/MIME

2.2.1 Key Elements

The key element of S/MIME are the certificates. The certificate is used for the communication with another user. The private key with the certificates are stored locally in a certificate store. Several different certificate stores exist on one device. To enable S/MIME communication with automated synchronisation in a multi-device environments all of these certificate stores need to be accessed and the necessary certificates must be selected and synchronised to all other devices.

2.2.2 Implementations

Other than OpenPGP, S/MIME is often implemented directly in the email applications on the devices, no additional installation is required in order to use S/MIME. The challenge to use S/MIME on multiple devices is to select all necessary S/MIME certificates, with that every certificate store must be searched for potential S/MIME certificates of communication partners.

2.2.3 Conclusion

With S/MIME being an additional specification that secures email communication, a difference can be seen between OpenPGP and S/MIME. But the same problems than with OpenPGP must be solved. The needed certificates must be selected of the different certificate stores and must be transferred to all devices. On these devices they have to be added to the local certificate stores on each device. For the private key the owners S/MIME certificate must be exported and the private key must be attached to it.

Chapter 3

Problem Analysis

3.1 Appearance

Securing emails with current mechanisms like OpenPGP [3] and S/MIME [2] works well, but the functionality is not available across multiple devices. 20 years ago — when these mechanisms were defined — the communication of a person based on the usage of just one device. But since then the communication has changed in a way that people don't use just one device anymore. People use their laptop, smartphone, tablet, or desktop PCs. On all these devices a user wants to access his complete email communication. Even though all emails get synchronised the user can just read the unsecured emails on every device. Emails that are secured by either OpenPGP or S/MIME can not be read as long as the necessary information, like private and public keys, are not available on the device they want to use. This problem is caused by the following reasons.

3.1.1 Multiple Devices

With multiple devices the necessary information of OpenPGP and S/MIME have to be available on each device. Therefore for an encryption or decryption required private and public keys need to be exchanged between all devices. Also trust information must be available to provide the required authenticity of the keys.

3.1.2 Mobile Devices

Mobile devices generate additional problems. Applications on this devices run is a sandbox mode. This causes problems of sharing the same data with different applications. The files can be saved on the local storage of the device but other applications can not access them automatically. They import the files to the local storage of each application.

3.1.3 Sending Mails

To be able to send emails between two communication partners on a device specific information must be available on that device. This information include the public key of the recipient and the private key of the sender. If the public key is not available an encryption of information is not possible. If the private key is missing the integrity of the sender can not be verified. Only if both element are available a secure communication available on that device.

3.1.4 Reading Mails

The problem for reading mails is similar to the problem of sending a mail. In order to do that also certain information must be available on the device that is used to read that email. These information include the private key of the recipient and the public key of the sender. The private key is necessary so that the mail can be decrypted and read by the user. The public key is required to verify the sender. If one element is missing the secure communication is again not available on that device.

3.1.5 Trust Information

A secure communication is based on trust information. For OpenPGP this trust for a public key is generated with a web of trust. For S/MIME central authorities are responsible to assign a public key to a user. If these trust information are missing the authenticity of public keys can not be guaranteed. This results in an additional signing of the keys. It is important to have all the trust information available on each device that is used for secure communication. With that the redundant task of verifying the public key is not necessary.

3.2 Approaches

A possible approach is to exchange the necessary information for OpenPGP and S/MIME between all devices. With that the communication is available everywhere and fits in a multi-device environment. In order to achieve that goal the private and public key of an user must be extracted form one device where the keys already exist. In addition to that all public keys of communication partners that are available must be extracted as well. For GPG also the trust database needs to be exchanged so the authenticity of the keys is available on all devices. But just to extract all information from the local storage or the certificate stores of a device is very complicated and non technical users are not able to perform this task. Besides this the user has to be on track about changes. So the user that wants to have all information on his devices he has to check for every change manually. Some changes are not recognisable and information can be easily lost when a file gets overwritten.

3.3 Conclusion

To enable OpenPGP and S/MIME in a multi-device environment all necessary information must be exchanged between all devices. This information include private keys and public keys of the user, as well as the public keys of all communication partners and trust information about this keys, like the trust database of GPG. A manually exchange is not possible, so an application must be created that detects related changes and synchronises them automatically to every device. This ensures simplicity so that every user, even non technical ones can use this application.

Chapter 4

Related Work

4.1 State of the Art

Currently different companies try to solve the problem of key management for secure communication in a multi-device environment. All the companies want to create an easy synchronisation of secure and public keys to enable this secure communication. They also pursue the goal that the user does not have to take care about the synchronisation of their own keys, but it will be handled by the companies. In this chapter these approaches will be introduced and evaluated.

4.1.1 Synchronisation of OpenPGP Keys

The company Whiteout[1] offered a web-based secure email service. They took care about all synchronisation so that the user could access the emails from everywhere. They provided a approach of a synchronisation of PGP keys. They created a way where an user can sign up for their service and upload his own keys to their server. So when the user uses their client all keys where available. The security aspect was guaranteed by a symmetric AES-256-GCM encryption [5]. This symmetric encryption is protected by a 24 digit random passcode. This passcode is never stored anywhere so it should be written down by the user. But after the setup on each device a secure email communication is possible. But the problem here is that a user has to give his keys to another provider. At least the user is able to work with his already existing PGP keys and does not have create a new key pair. Even the communication with other users outside of Whiteout is possible because the email encryption is based on OpenPGP Specification [6]. So every user of Whiteout can also send mails to their friends without Whiteout. But the user is still forced to user the email client given from Whiteout to access their keys. So the user is not independent from the company and in an environment with such sensitive

[1]https://www.whiteout.io

data that is a big problem. Never the less the attempt is pretty good and if it is extended in the right way it offers a good foundation for a new attempt.

4.1.2 Secure Communication in Closed Systems

Other to Whiteout there are some more approaches to enable secure communication with multiple devices. But those approaches are handled in closed systems with own implementations of OpenPGP, like Protonmail[2]. This company provides a service which allows secure communication out of the box. But with Protonmail it is not possible to communicate with people who are using OpenPGP with their own mail client. [7] Again the user is forced to use the applications given by a certain company. The difference between Protonmail and Whiteout is that Protonmail is completely closed. The user is not able to import his old keys but Protonmail creates new ones for them. Protonmail does everything for the user so he can start sending secure mails without having to take care about his keys or anything else. But it is not compatible to other services. This attempt of key synchronisation is completely focused on simplicity in combination with high security. But the trade off between simplicity and restriction with Protonmail is very hard. So every new user starts completely from scratch and has to create a new email address and a new key pair.

4.2 Security Tools

In addition to the existing specifications like OpenPGP or S/MIME and current approaches there is another topic which has to be mentioned. This topic contains already existing security tools. These tools cover different areas of usage. But they target on one goal: to improve the security of a users information. The tool that is presented is a password manager. This tool is a good example how data can be gathered on a local device. And that data is secured by the application so that the content is save and exchangeable between several computers. These information are important for the requirements because the synchronisation application will adapt some aspects and combines them to offer new functionality.

4.2.1 KeePass

With KeePass[3] the user can store his passwords in one secure place. Inside the application the user must select one master password. With this password the user can access the key database that is created by the application. After selecting a master password

[2]https://www.protonmail.com/
[3]http://www.keepass.info

the user can easily add all login data he has to remember and then they are stored in just one file which is just accessible with the selected master password. The security is provided by a symmetric encryption with AES [8]. This shows how it is possible to create an application where user add very sensitive data which is then protected. It is also important to see how symmetric encryption on client side can secure information and protect them from manipulation of others. The problem for KeePass is that it also does not have a multi-device support. So to use KeePass on multiple devices the user has to synchronsie the file by him self.

4.3 Summary

In this chapter different very important information where presented, which are essential for a usage of OpenPGP [3] or S/MIME [2] with multiple devices. In addition to that some very interesting attempts could be offered which are highly adaptable. But also the security tool provides a new point of view how a synchronisation application can work in the end. With all the information gathered in this topic the requirements for the secure synchronisation tool is much easier to define. Due a first preview the requirements can focus on the important information and the attempts that offer a good pattern to build on.

Each of these approaches target the problem of key management directly. But each company combines it with own communication applications. So the user has to use the application that are provided by the companies and so they are completely integrated in their system. But to enable OpenPGP and S/MIME as they already exist in a multi-device environment another approach must be created. An approach that is not bound to one communication system like Protonmail. It should be independently usable.

Chapter 5

Requirements

With the existing problem analysis and the problems of the current approaches, the requirements of a new approach can be defined. The approach in this thesis is an application with an automated synchronisation mechanisms, that should work along with the securing mechanisms OpenPGP and S/MIME. To define the requirements, the stakeholders have to be analysed and use cases must be created in which such an application can be used. These stakeholders and use cases depend on the environment in which the application will run later on. For an usage inside an organisation, the organisation will perform as a stakeholder and use cases for an administrator must be created. For a single end user this stakeholder and use cases are irrelevant.

5.1 Stakeholder Analysis

As stakeholders the end user and the organisation appears. The end user as a person that synchronises his data and the organisation that administrates a synchronisation server.

5.1.1 End User as Stakeholder

An end user wants to synchronise all key material that belongs to him. Wether it is S/MIME or OpenPGP. The user doesn't want to get involved in the synchronisation and doesn't want to perform many actions with the application, because it should not be required to have any knowledge. With that even non technical persons can use this application. For the end user it is essential that all his key material is on the same state on every of his devices.

5.1.2 Organisation as Stakeholder

In an organisation, administrators will be able to monitor the access of the user to their keys and have control over the users that can use the system. Additionally the administrator will also have the control over the number of users and the control which users can access their key material and who doesn't. The administrator does not care about the content of the key material. They still belong to the user. The administrator just wants to restrict the access if a user leaves the organisation. With that the access to information that belongs to the organisation can be limited or prevented.

5.2 Use Case Analysis

The use case analysis is divided like the stakeholders. Because every persons has other interests in the system and wants to do other things with it.

5.2.1 Use Cases for an End User

5.2.1.1 Installation

For a user there are several possible motivations for installing the application. The user installs the application on his first device. He does not have any keys in the beginning, so no keys must be synchronised. The other option is that the user already used OpenPGP or S/MIME before and has an existing key storage. In that particular case the user wants to select the key material and the application should take care about uploading his existing keys to the synchronisation mechanism and that they are accessible for other devices. After he installed the application for the first time on one device the user wants to install it on another one. So the synchronisation mechanism must detect that there are already keys available and has to download them to the new device.

5.2.1.2 Synchronisation

Before the synchronisation can happen, the user wants to select the mechanisms, like a dedicated server or cloud storage, that is used to synchronised the information. After that the user does not want to be involved in the synchronisation and the application should handle the rest.

5.2.1.3 Configuration

When the user deletes a device the key material should not be lost or deleted on any other device. But the user wants to synchronise his additions, deletion, or changes immediately. In addition to that an user wants to add new securing mechanisms over the time, so the amount of information that needs to be synchronised increases over time.

5.2.2 Use Cases for an Administrator

The administrator wants to organise the backend system in an organisation. So the synchronisation mechanism that will be used by the end user is a dedicated server. The administrator wants control over the user and the access to the key material. He wants to add new users oder delete users if necessary. He wants to be able to disable and enable users on the server. With deletion or deactivation the user will lose all rights to synchronise their key material. This requirements are important due the special requirements of an organisation that needs control over the users and wants to host the server by them own. These requirements are necessary when a new person enters or leaves the organisation.

5.3 Functional Requirements

5.3.1 Synchronisation

The application must be able to synchronise information. This information need to be uploaded, downloaded and exchanged with the local file. The files must be selected just once. The application has to save the path to the selected files. After the selection these files have to be synchronised until the user disables or deletes them.

5.3.2 Consistency

Consistency is important because no information must be lost at anytime. So a version version control is necessary in order to provide this consistency. Every application must be able to compare his current version with other ones an in case of conflicts the application must be able to merge several different versions to one new version without any dataloss.

5.3.3 Automation

With automation, less user interference is required to use the synchronisation mechanisms. This means changes must be detected automatically and all task relating to the synchronisation must be executed.

5.3.4 Encryption

Encryption of the information that is synchronised is essential. The content of these information, like private keys and trust information, are very sensitive. Before they can be exchanged between devices or stored in a storage backend, these information need to be encrypted. As well the information of the files that have to be encrypted the connection between two exchange elements, like client and server, must be secured as well, e.g. with TLS.

5.3.5 Configuration

In addition to the functionality that is directly connected to the synchronisation the application must provide more functionality for configuring the application. A synchronisation mechanisms must be selectable and all different securing mechanisms like OpenPGP and S/MIME must be selectable as well. This is necessary in order for the user to enable different mechanisms that he needs and wants to use.

5.3.6 Backend

The backend needs to authenticate a user when he enters the system. This is required in order to match saved key material to a certain user. In addition to the authentication the backend has to provide an interface for exchanging the data. This interface must provide an upload and download functionality, as well as metadata of the information.

5.4 Technical Requirements

5.4.1 Frontend

The technical requirements for the frontend contain the need of an extensible application. With such an application it can be ensured that new securing and synchronising mechanisms can be added easily at any time. This is supported by reusable code and between each exchangeable element of the application an existing API should exist. With such a reusable code the maintenance can be reduced as well. In addition to the extensibility the application should work with the less user interference. But to enable that the data integrity must be ensured so no data is lost at any time. As well to the APIs inside the application the API to communicate with a synchronisation mechanisms must be exchangeable as well. In order for the application to work correctly the access to OpenPGP and S/MIME information must be enabled.

5.4.2 Backend

The backend mechanisms needs to be exchangeable, but has to provide a necessary API which fulfil the functional requirements. As well as the exchangeability the backend is not allowed to access any information and is just used for storing the files. Additional requirements for the administration can emerge later on.

5.5 Design Requirements

The application must run on many platforms, because user have not just different devices, they also use several different operating systems. In order to enable the usability for as many people as possible the most common operating systems must be supported. This includes MacOS, Windows, Linux, Android, and iOS. In addition to the multi-platform support a simplicity must be provide, so that every kind of user can perform with this application. To provide this the application needs clear steps and no user knowledge must be required. For the design an important factor is security. The information that needs to be exchanged is very sensitive. So every step that these information takes, needs to be secured in a way that no manipulation is possible at anytime.

5.6 Summary

With all the requirements fulfilled the following feature list is enabled for the synchronisation system:

- Available on multiple operating systems
- Synchronisation of sensitive data for OpenPGP and S/MIME
- Automated Exchange of data
- Extensibility of the application
- Security for the data and the connection
- Consistency of information
- Exchangeable backend

Chapter 6

Design

6.1 System

6.1.1 Description

The system is divided in two different parts. One part is the application for the client. This application initialises the synchronisation, is responsible for exchanging the information locally for the user, and transfers the information to the backend. The backend is the second part. This part is responsible for storing the data and distributing it to every client that wants to access the data. The information that are stored on the backend are assignable to exactly one user.

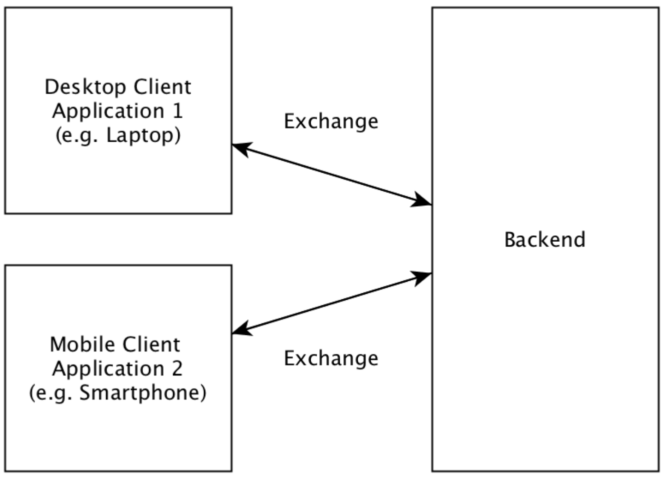

Figure 6.1: Diagram of the system

6.1.2 Functionality

The system can exchange information between different devices. This information include sensitive data for OpenPGP and S/MIME. These data consists of public and private keys and trust information. The Consistency is guaranteed by the system, because a version code for each information is created when the information enters the system. This version code is changed when the information is changed and with this change the system can compare different versions and distribute the correct one to every client. The information is secured inside the application with a symmetric encryption and during the exchange, the connection to the backend storage must be secured as well.

6.2 Backend

For the system to run different backend mechanisms exist. The difference between these mechanisms is how the data is saved to be accessible for other devices. Each backend is completely independent from each other.

6.2.1 Server

The server is a dedicated server that is created just for the synchronisation system. This backend mechanisms is useful for organisations, like TUM, because with an own implementation the requirements of the organisation can be fulfilled and the system can be administrated. It provides a defined API and with that API it is able to standardise the communication between the server and the application.

6.2.1.1 Functionality

The server authenticates users by a username and a password. After the authentication users can access their data and synchronise their information on several devices. In addition to the user functionality a dedicated server can also fulfil administrative requirements and an administrator can manage the access of the user for the system. The functionality of the server can be extended with the functionality that the organisation needs. Only the synchronisation API must be available for the application.

6.2.1.2 Structure and Interfaces

The server is divided in two parts. One part is responsible for the communication with the application. In this part a REST API is implemented and triggers every other functionality of the server. This REST API consists of HTTP requests so the data is transferred from and to the server with GET, PUT, POST, and DELETE requests. The other part is the persistent storage. This is represented by a database and server storage where the files of the user can be saved. Both of these parts communicate with a interface, that enables the access to the database and the server storage for the communication part. Other additions in the structure are also available as long as the required REST API for the application is provided and the data is exchanged correctly.

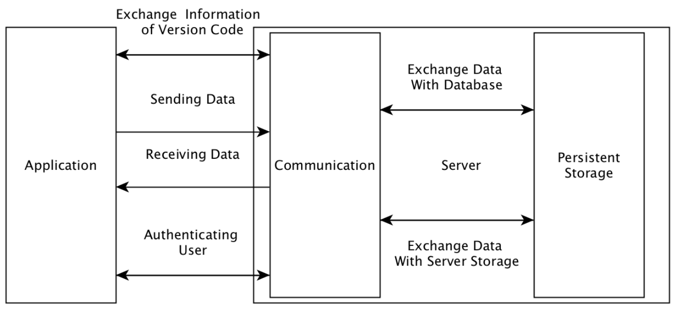
Exchange Information of Version Code
Sending Data
Receiving Data
Authenticating User
Application
Communication
Exchange Data With Database
Server
Exchange Data With Server Storage
Persistent Storage

Figure 6.2: Server structure

6.2.2 Cloud Storage and Peer-to-Peer

In addition to the dedicated server other backend mechanisms can be used. These mechanisms target more a single user that uses the application without an organisation.
Cloud storage: Cloud storages, like Dropbox offer a file API. That API can be used by the application and file can be synchronised with the cloud storage. In order to do that the user has to authenticate himself with the application. The data can not be manipulated by the cloud provider, because all information are encrypted on client side before they are uploaded to any backend storage. The option to administrate different users is not possible with a cloud storage backend, because no administrator is available. A persistent storage like a database is also not available. This causes the need that all information about version codes must be save inside of files that are also uploaded to the cloud storage.
The cloud storage enables an easy synchronisation mechanism for everyone with an account for the certain cloud storage.
Peer-to-peer: Peer-to-peer does not require a backend like a server or cloud storage does. With peer-to-peer the information get synchronised directly between the devices. This results in some changes how the application works. The client of the system contains the functionality of a peer-to-peer exchange.

6.3 Client side

The application runs in two different environments. One is the desktop environment, where the application has access to the local storage and is not limited to any sandbox. The other one is the mobile environment, where the application has restricted access to the local storage. This results in some differences in functionality on each platform, as it is not available in both environments.
In addition to that the application is extensible. Because of that attribute, the application itself is divided in three modules. One of these three modules is the core module. This module is responsible for controlling the application. Inside the module the application manages the other two modules and the main user interface. The other two modules, one for gathering and exchanging the files on the local storage and one for connecting and exchanging the files with the storage backend, are exchangeable. That means, the functionality depends on the information that need to be synchronised and how they are synchronised. Both modules are based on plugins to simplify this exchange.

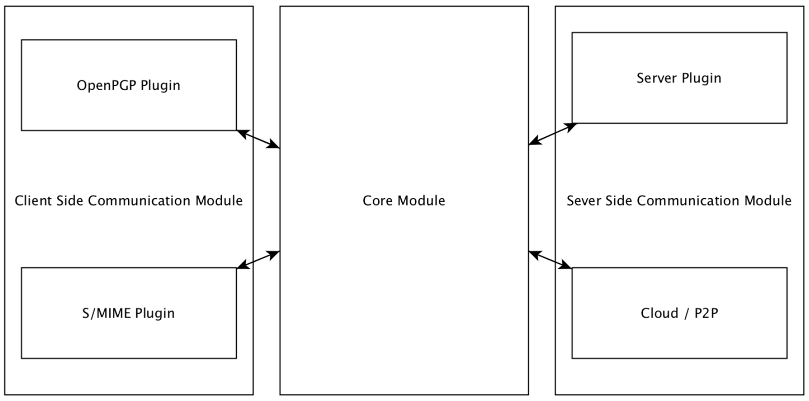
OpenPGP Plugin
Client Side Communication Module
S/MIME Plugin
Core Module
Server Plugin
Sever Side Communication Module
Cloud / P2P

Figure 6.3: Diagram of the application

6.3.1 Core Module

The core module is the hearth of the application. This is where main the user interface is created and the main logic is inside. The main functionality of this module is to gather all data that is provided by the client side communication modules. The client communication passes the data that they read from the hard drive to the core module. Here the data gets encrypted and the data get passed to the selected synchronisation module that the application uses. So the core module is not responsible for any comparison of the data if it changed but it is responsible for securing the data for the rest of the synchronisation. The user interface contains pages that the user can interfere with and the core system is responsible to display the current state of the application, like are all files up to date, or what other plugins and modules are installed.

6.3.1.1 Functionality

The functionality of the core system is completely generic. This module does not care about the file that are passed to it, but is responsible that all information is secured. This module is also has to take care about the availability of all necessary functionality for all other plugins. So it loads the configuration files and prepares all used frameworks, so that they can be used in every plugin individually. Other than that the rest of the functionality is handled by other modules which are specialised for that. But the modules are connected to the core module with an interface that enables the necessary communication between the core module and the other two.

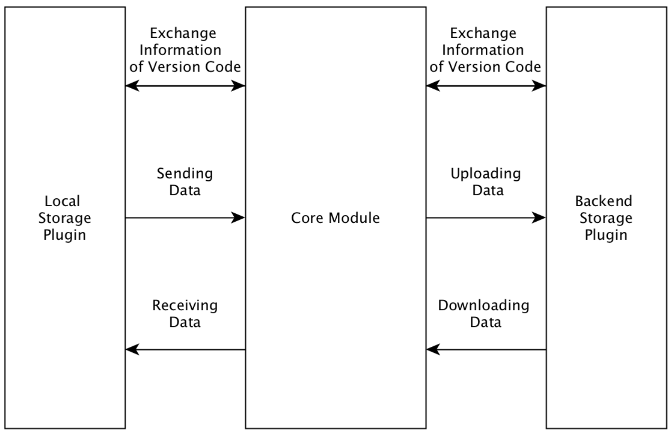

Figure 6.4: Integration of core module

6.3.1.2 User Interface

The user interface of the application is reduced because a user should not have to interact a lot with the application it self because it runs mainly automated. But the user interface should contain pages for showing the user information that he may needs, like are all keys up to date or a button to update the files manually. The Core system also has to display current plugins that are active in the application and a link to get to the correct user interface of the plugin. The rest of the application is running mainly without the interference of the user so no user interface is needed there.

6.3.2 Plugin System

The whole application is divided in three modules with contain certain functionality. Two of these modules are exchangeable and based on plugins. These plugin are specialised on certain functionality which is responsible for exchanging the files locally or exchanging the file with the backend storage. With this plugin system the functionality of the application can easily exchange or extended, because existing plugins can be added or deleted.

6.3.2.1 Local Storage Plugin

Functionality: These plugins are specialised to take information from the local storage and exchange the information with new ones. Each plugin is responsible for exactly one information type. These information type defines what data has to be synchronised in order to enable the intended functionality. Right now this types are OpenPGP and S/MIME. For each information type certain functions are necessary to gather information or exchange it with the local storage again. Besides the modification of information, this plugin is also responsible for enabling the consistency of the information. This is provided by the decision if information needs to be downloaded or uploaded. In case of inconsistency these plugins have to solve the merging problems. These problems can differ between the information types so this functionality must be implemented separately for each plugin.
User interface: In addition to the functionality each plugin offers a user interface, because every information type has different dependencies how they can access the information on the local storage. These dependencies can be set up in the settings interface of each plugin. These settings include the information source, the current version of these information and the functionality of manual exchange.
Integration: These local storage plugins are connected to the core module with a fixed API. This API enables the exchangeability of the plugins. The functionality of this API includes giving information to the core module or get information of the core module. Everything that is done with the information is not in charge of the local storage plugins. The core module is completely responsible for that. The API includes the functionality of exchanging current version code information too.

6.3.2.2 Backend Storage Plugin

Functionality: The backend storage plugins are responsible for exchanging the information with the storage backend that saves the information outside of the local device. Each different type of backend storage plugins is specialised for on type of storage backend. Right now this types can be a dedicated server or a cloud storage. For a peer-to-peer exchange the plugin handles the exchange different. Here is no external backend storage required, but a direct connection to the other devices.
User interface: The user interface of each backend storage plugin contains settings for the exchange platform. For a dedicated server this includes the server domain and for the cloud storage this includes the credentials. No domain for the cloud storage is

required because every provider needs a own plugin and the requirements for accessing the cloud API are different.
Integration: The backend storage plugins communicate with a given API with the core module. For these plugins the content of the information and the used local storage plugin are irrelevant . The determined API include functions, like giving data to and getting data of the core module. In addition to that the API offers the function of information exchange about the current version code.

6.3.3 Client Side Communication Module

The client side communication module is the part where the local storage is accessed. To do that it is using the plugin system as mentioned before. Because the application is about enabling secure email communication, the architecture considers two different securing mechanisms, OpenPGP and S/MIME. But because it is plugin based other securing mechanisms can be added as well even when the first implementation of the application is finished. To be fully integrated these plugins must be working like the local storage plugins that where defined earlier.

6.3.3.1 OpenPGP

The OpenPGP plugin is specialised for the GPG implementation of OpenPGP. This implementation, as defined in Chapter 2, organises the required information in two or three files on the local storage. The amount of file is depended on the version of GPG that is used.
Functionality: The plugin has the knowledge about where these information are stored. This knowledge is predefined by the standard storage folder for these files or the custom settings by the user. With this custom settings the user can select where the files are stored if the file path differs from the standard one.
The knowledge about the file path is necessary because the plugin can use this information to observe the necessary GPG files. With this observation the application is notified if one of the file has changed and an update is necessary. With this notification the plugin sends a request to the core module for the current version code information of the files on the storage backend. With this information the application can detect if other changes are saved on the backend that are not already on the local storage. If the local version is the newest version, the plugin selects the modified file and gives it to the core module to enable the upload.
But if there are new changes on the storage backend, that are not in the local storage, the existing files are downloaded from the backend storage and the files are merged. This is possible for the public keys but not for the private keys if GPG version less than 2.1 is used. [4] After a successful merge the new file can be selected from the local storage and can be given to the core module to upload it to the backend storage.

User interface: The user interface provides the information about the current file path of the GPG files and the local version code information. In addition to the information, the user interface enables the functionality to change the file path of the GPG files and to initialise a manual synchronisation process.
Integration: As mentioned in the functionality of the OpenPGP plugin, the plugin is integrated to the application with a fixed interface to call certain functionality of the core module. An interface for the information about the current files and for uploading and downloading files must be provided. With this integration it fulfils the requirements for a local storage plugin.

6.3.3.2 S/MIME

The S/MIME plugin is specialised on the usage of certificates which is required for S/MIME. That means the plugin is responsible for collecting all necessary certificates.
Functionality: For this plugin to work properly, it needs access to certificates storages. In this certificate stores all necessary certificates are stored. With this access the plugin can gather these certificates and a synchronisation is possible.
To synchronise all S/MIME related certificates the stores must be browsed and the certificates must be extracted. When this is done the S/MIME plugin must package these certificates and give this package to the core module to upload the information. When certificates are downloaded because of a change or addition. The S/MIME plugin receives them from the core module. These new certificates need to be inserted to the right certificate store. But before the plugin can upload the package of S/MIME certificates the local version code must be compared to the version code on the backend storage. When the local version code is newer the plugin passes the information to the core module. But when a conflict exists both packages need to be compared and merged to a new package. This package is then uploaded and the missing certificates are added to the local certificate store.
User interface: The user interface offers the option to add or exclude certificate stores. With that the application knows what certificate stores have to be browsed. In addition to the list of certificate stores, the user interface also shows information about the current version and the package of S/MIME certificates. The user interface offers a manual synchronisation function too.
Integration: The S/MIME plugin is integrated as every local storage plugin. The plugin is connected to the core module by requesting information about the current version code of the backend storage and by passing data to and receiving data of the core module.

6.3.4 Synchronisation Side Communication Modules

The synchronisation side communication module is the part where the application connects to a backend storage and exchanges information and data between backend and client. Because there are several different backend mechanisms, like a dedicated server, cloud storage, or a P2P connection, this module is using the plugin system, mentioned before. This module gets data from and gives data to the core module.

6.3.4.1 Server

The server plugin is specialised to work with a dedicated server that was created for the application. So this server provides a necessary REST API that the application can connect to.
Functionality: The plugin can authenticate the user with the dedicated server. This is necessary because the data that the application uploads must be assignable to exactly one user. When the user is authenticated, the plugin receives data from the core module and then the plugin connects to the dedicated server and uploads the data. When the core system requests data, the plugin connects to the dedicated server and downloads the requested data. In addition to the up and download functionality the server plugin provides a functionality to get the current version code information of a file from the server. Also this is requested by the core module.
User interface: The user interface enables the user to set up the server connection, by entering the domain or IP-address of the server. This is required so that the application can connect to the correct server.
Integration: The integration to the whole application happens by the interface of the core module. The server plugin passes data to or receives data from the core module. This interface also enables the core module to demand for backend information of the version code of certain data.

6.3.4.2 Different Synchronisation Plugins

Beside the connection to a dedicated server, which was created for the synchronisation with the application, the application also offers different synchronisation mechanisms. Because of the difference between these mechanisms they are divided in different plugins.
Cloud storage plugin: Cloud storages, like Dropbox, offer a file API. After authenticating the user for this API, it can be used to upload and download file to the cloud storage. This plugin is integrated to the core module with the same interfaces than the server plugin. This means that the core module can request information about the current version code of the data and request a upload and download of files. Every

request of the core module is handled with the file API of the cloud storage. For every information about the current version code the application creates a file and uploads it. This is caused by the missing database of the cloud storage. The plugin must provide the necessary functionality to authenticate a user. For that the user has to enter his credentials in the user interface for the plugin.
Each cloud storage needs their own implementation because the provided APIs of the cloud providers are different and different tasks need to be executed.
Peer-to-peer plugin: The plugin for peer-to-peer avoids a backend storage completely. So the plugin has a different design then the plugins of a server or cloud storage. To enable peer-to-peer, the application of one device must connect directly to the application on another device. With this mechanisms both versions must be compared by one application. After that comparison the new data must be provided to both devices.

Chapter 7

Development Life Cycle

There are a number of possible implementations to fulfil the requirement that the application will run on as many platforms as possible. But for the implementation also the maintenance is very important. So when the application is finished the amount of time to fix bugs or to extend the application have to be reduced. Either way there are three possible solution for the implementation that were investigated. One solution is to implement each client within its own native environment, the next possible solution is to split the application in a mobile and a desktop application, and the last solution is to implement the application with a hybrid codebase that runs on every device wether it is mobile or desktop. The one that was chosen can fit all requirements and is the easiest to maintain on so many platforms.

7.1 Native Implementations

With separate implementations for each platform a native code environment is created. With specific behaviour of each environment, those behaviours can be used to create an optimised code. But if the implementation is completely separated between the platforms it takes a lot of effort to create a stable application for each device, because the design and the requirements must be fulfilled for every platform individually. In addition to the this effort, the maintenance after finishing the application is very high. Bugs that affect every platform need to be identified and fixed individually as well. This is very time consuming. Along with the maintenance the extensibility is an important factor of the application. In order to offer such an extensibility each platform needs to extended. That takes a lot of time. The last point that needs to be mentioned is the man power that is necessary to have separate implementations. For just one person the implementation will take very long because no consisting codebase can be used across the implementations and a developer needs a lot of knowledge about each native language like, C#, Obective-C, or Android-Java just to mention a few.

7.2 Separation between Mobile and Desktop

The way how applications run in a desktop and a mobile environment is different. On desktop devices an application has complete access on the local storage and so an observation of files is possible. On mobile devices an application runs in a sandbox. This means mobile devices do not share their local storage like in a desktop environment. The applications can save data to the local storage but it is not automatically accessible for other applications. Because of that a separation of mobile and desktop can be an approach.
For the desktop application Java would fit the needs of a multi platform programming language. But one problem of Java even when the application is done, it is required to have Java installed on every device that wants to use the application later on. This can be complicated in organisation where users are restricted to a certain Java version or no Java installation is available. Even with a restriction of a Java version, the application has to use the newest version that is available on all devices. This can lead to a Java version which is several years old.
The mobile application can be implemented in two ways. On way is to have again separate implementations on the mobile side. But now also two additional implementations need to be created and this takes again a lot of effort. The other approach is to have a hybrid mobile application. So just one implementation is necessary to fit iOS and Android. For such an approach PhoneGap[1] is an option. With PhoneGap it is possible to create a web application that runs as a native app with the access to the device like a native one. With this approach a lot of effort can be saved because just one implementation is necessary to fit in both mobile operating systems.
To summary that approach, several implementations are required to fulfil the goal to run on multiple platforms. The Design must be implemented in at least two different ways. Also the maintenance and the extensibility is split.

7.3 Hybrid Implementation

The last option that was investigated was to implement the application on every platform as a hybrid application that is based on a web application. Similar to the mobile approach with PhoneGap[2] this solution can also fit in the environment of desktop devices. The framework Electron[3] does fulfil the needs for that. So with PhoneGap and Electron it is possible to have the same web application as a native one on desktop and mobile devices. With that approach, it is possible to have the same codebase for every device, so the maintenance effort is much lower than for the other ways of implementation.

[1]http://www.phonegap.com/
[2]http://www.phonegap.com/
[3]http://electron.atom.io/

With the usage of an web application other problems appear, because web application often appear as one. A feeling of a real application is not created when every page gets loaded individually. In addition to the look and feel of the application there is a problem of combine the functionality of PhoneGap and Electron. They both offer similar functionality but how the functionality is provided is completely different. Electron uses NodeJS as an additional framework inside to offer certain functionality, like accessing the local storage of an user. PhoneGap enables such functionality completely different by offering APIs for those actions. Even when the codebase is similar there are still differences in the code how it has to be executed. A differentiation inside the code is necessary to bring all functionality to every device.
The implementation is also completely different than an implementation with native languages, so finding bugs can take more effort. But these fixes just need to be applied once because the whole application shares the new code. The extensibility is also provided with the help of the same codebase. Each new functionality or mechanisms needs to be implemented once. Just the difference between PhoneGap and Electron must be considered.

7.4 Conclusion

	Multi platform	**Maintenance**	**Automation**	**Security**
Native	yes, but multiple implementations		yes	yes
Separation	yes, but two implementations		yes	yes
Hybrid	yes, same codebase		yes	yes

Table 7.1: Comparison of implementations

Even though all attempts of an implementation can fulfil the requirements, the approach with the least maintenance effort must be selected, in order to be able to create such an application quickly and maintain it when it is finished. Because of that, the hybrid approach is chosen. The other approaches have to many negative sites like the amount of time that is required for maintenance or the dependence on certain versions of a programming language, like Java.

7.5 Creation of a Development Life Circle

To fulfil the needs of a native application the existing problems must be solved by the usage of different frameworks, so that the web application has the right look and feel, but also enables the required functionality for the different platforms. A good working development environment is in a hybrid approach very important. Especially when different frameworks are combined to one project.

7.5.1 Suggested Frameworks

The frameworks that are suggested below fulfil certain needs that the application has to solve in the hybrid approach. Each of these frameworks target on one problem and solves it in a way that the end product is an application that offers every needed functionality.

7.5.1.1 Electron

Electron[4] is a framework that allows to compile web applications as native desktop applications. With this one framework the application can run on MacOS, Windows, and Linux. Electron allows the usage of all common web technologies like HTML5, CSS3, and Javascript. This is possible because the Chromium browser engine provides everything that is needed to execute those three technologies. In addition to the normal web technologies NodeJS can be used as well. So with the help of this framework it is possible to create applications that can perform every necessary actions directly on the device, like reading or writing files, which is not possible with normal Javascript.

7.5.1.2 NodeJS

NodeJS[5] is normally used as a language to create Javascript based backend servers. With the integration of Electron the complete functionality of NodeJS is also available inside of the application. So the app has a file path, can access the hard drive, and enables the usage of the newest Javascript engines which are not available in normals browsers yet. NodeJS in general also allows the "Node Package Manager"[6]. This is used to install all the packages like Electron, PhoneGap, or EmberJS. It enables automatic update when certain commands are run.

7.5.1.3 PhoneGap

PhoneGap[7] can be compared to Electron but it fulfils the needs of bringing the application to mobile devices. Electron is restricted to desktop application, so PhoneGap has to fulfil the requirements of mobile devices. PhoneGap transforms normal web applications to native ones, so with that tool the codebase is transformed to an Android or an iOS application. A positive factor of PhoneGap is that it also grants access to local device actions, like saving files. This is for an application that synchronises data essential.

[4]https://electron.atom.io/
[5]https://www.nodejs.org/
[6]https://www.npmjs.com/
[7]http://www.phonegap.com/

7.5.1.4 EmberJS

The tool listed above enhance normal web applications and transforms them to native applications with rich functionalities. But the problem of a web application is that it feels like an homepage if the page gets loaded when a button is clicked. For that problem EmberJS[8] is the solution. That framework allows full web applications with is powered completely by Javascript and enables a one page application. This is possible because EmberJS renders every page of the application to one website that is loaded in the beginning. So a feeling of real application is created. EmberJS also takes advantages of the MVC model. So user interface and logic is completely separated. This is really comfortable to create or maintain an application. With EmberJS are very clean coding structure is given from the beginning on and a clean coded application can be created easily.

7.5.1.5 Grunt

To use all of the different tools a lot of different command line tasks is necessary to run the application during development or building it to release it. Grunt[9] is a framework that gathers all commands and creates new ones which executes more than just one task. So the usage of the command line becomes much more cleaner and easier to run and build the application. In addition to this no necessary commands to start up the application correctly can be forgotten. It improves the over all workflow during and after the development and the maintenance phase.

7.5.1.6 Other Javascript frameworks

Some more frameworks are used inside of the code. ForgeJS[10] is another framework. It is a Javascript implementation of the common encryption standards like RSA, AES, or TLS. With that framework it is possible to secure all data inside outside of the application. This encryption framework is widely used and offers a rich functionality of encryption mechanisms.

7.5.2 Conclusion

All frameworks combined result in the development life circle. The user executes Grunt commands to use every framework. Grunt triggers actions for Electron, PhoneGap, and EmberJS. Since EmberJS is running inside Electron or PhoneGap these frameworks

[8]http://www.emberjs.com/
[9]http://www.gruntjs.com/
[10]https://www.github.com/digitalbazaar/forge

grant access to different native functionalities. In addition to these native functionalities, EmberJS is also extended the framework ForgeJS for encryption. EmberJS provides the content of the application for Electron and PhoneGap. Electron creates the applications for Windows, Linux, and Max. PhoneGap creates the applications for Android and iOS.

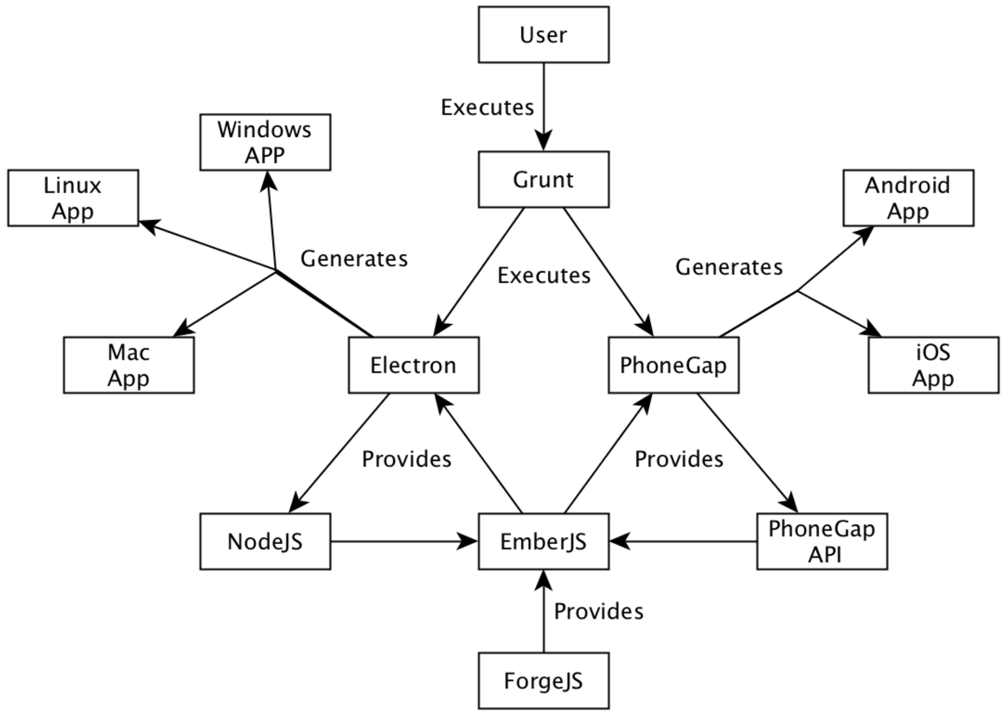

Figure 7.1: Diagram of development life cycle

Chapter 8

Implementation

The implementation of the thesis project is completely based on the technologies that where evaluated in Chapter 7 and the design that was defined in Chapter 6. Therefore the implementation is split in the application and the backend. The implementation of the backend is represented by a dedicated server, so the approach of cloud storages or peer-to-peer connections are not implemented yet. The application uses the suggested frameworks. This means the application is written with EmberJS and is executed by Electron and PhoneGap.

8.1 Backend

The backend represents a example server. This server provides the necessary API that was defined in Chapter 6. With this API the application is able to transfer data to the server and receive data from the server. In addition to this the server allows the user to authenticate.
The server is implemented with a Java Tomcat server and the API is build with the Spring framework[1]. The functionality of the server is the minimum that is required in order to enable the system to run. Any administrative operations, like organising the users, need to be implemented. As the persistent storage an MySQL database is used and all incoming files are save on the local storage of the dedicated server.
The functionality of the server covers uploading and downloading files, checking the server for updates and authenticate and create users.

[1]https://www.spring.io/

8.2 Client

For the design in Chapter 6 a separation between the functionality of desktop and mobile environment is suggested because functionality of these two environments are different in the way how they can access the local storage of a device and the accessibility of the data for other applications. In addition to that the separation is necessary because the suggested frameworks Electron and PhoneGap just support either desktop or mobile environments. Every necessary command that needs to be executed is managed by Grunt to it easy for the developer to start the application for a certain platform

8.2.1 Desktop Environment

The desktop environment of the application is enabled by Electron. This framework provides the environment for the application itself to be executed natively on each device. Inside the Electron implementation necessary settings are made. But the content that is shown is generated by the application framework EmberJS. Electron expands the functionality of EmberJS by providing NodeJS as a functional framework.
The important files for the Electron framework are inside the Electron folder of the implementation directory. Inside this folder the most important file is "main.js". In this file the configurations are made. These configurations include window size, exit methods, the functionality of the menu bar, and the directory of the content of the application. For this application the content is inside the "www" folder. The "dist" folder contains all created native applications for Windows, Linux, and MacOS. The file "package.json" is the dependency file of node package manager. The necessary dependency include "electron-prebuilt" and "electron-packager". The package "electron-prebuilt" allows to start the electron app inside a existing Electron framework. The configurations are not the same as they are defined in the "main.js" file. It is used to test the application easily. The package "electron-packager" is responsible for creating the native applications for each platform. The different commands that are required for that are managed by Grunt.

8.2.2 Mobile Environment

The mobile environment of the application is enabled by PhoneGap. This framework is responsible for creating native applications for the mobile environment. Inside of PhoneGap the necessary settings for the platforms is defined, but the content that the application shows on the device is created by EmberJS. PhoneGap also expands the functionality of EmberJS by providing a special API to access native functionalities of the mobile device.
The important files for PhoneGap are inside the PhoneGap folder of the implementation

directory. Inside this folder the configuration is happening in the "config.xml" file. Inside this file configures each platform that the application is running on. The minimum Android version and all permissions that the application needs can be defined here. the path to the application is standardised. The application loads the "index.html" in the "www" folder. The EmberJS application is to big to be loaded during the start. So the "index.html" is an empty page and it loads the real application after the start is finished. This causes no delay during the startup but prevents crashes caused by the loading time. The content of the application is in the directory "www/app/". The generated applications are stored in the folder "platforms" and inside this folder each platform has an own directory.

8.2.3 Application Environment

The implementation of the application is realised by EmberJS and is stored in the folder "SecureSync". The "Gruntfile.js" is also stored here. That file is responsible for executing the commands with Grunt. The actual implementation is placed in the subfolder "app".

8.2.3.1 Configuration

Several different configurations and dependencies are necessary for EmberJS. The first dependencies are store in the "package.json" file. This file stores all node package manager dependencies. This includes several Grunt dependencies, like "grunt" or "grunt-shell" and Ember-CLI[2] dependencies. In addition to the node package manager Ember also uses Bower[3] for dependencies. These dependencies are stored in the "bower.json" file. With Bower frameworks, like jQuery[4] or Bootstrap[5] are imported.
The Configuration of the application happens in the "ember-cli-build.js". In this file frameworks, like ForgeJS and Bootstrap are imported. In the folder "public" another configuration file is placed. Inside this folder exists the file "node.js". This file imports all necessary dependencies of NodeJS into the EmberJS application. With that import these functions are available globally through out the application. In addition import of NodeJS, this file also creates or load the "settings.json" file. This file contains important settings for the application.

8.2.3.2 Core module

The core module is defined in Chapter 6 and is the main part of the application. The main user interface is implemented here. Every folder inside the "app" folder except for the

[2]https://www.ember-cli.com/
[3]http://www.bower.io/
[4]https://www.jquery.com/
[5]http://www.getbootstrap.com/

"app/plugin/" folder belongs to the core module. The encryption of the data is handled by "app/helpers/encryption-utility.js". In this file the information are symmetric encrypted. The different synchronisation mechanisms are handled by "app/helpers/synchronisation-utility.js". This file provides the necessary interfaces between the core module and the local storage plugins. The directory "application" contains the main page with a menu for the application. The authentication is contained in the directory "authenticated". Every "route.js" file is using an authenticated route which is created by the directory "authenticated". The file "route.js" inside this directory checks if the user is authenticated. If the user is authenticated the requested page is shown. If not a login form is shown.
The automation of the application is enabled by the functionality of observing the required files for the plugins. These files which need to be observed are saved to the "setting.json" file. And every file that was added there is observed by the application. Now when a file has changed this file automatically triggers the process of synchronisation.

8.2.3.3 Plugins

Right now the implementation contains two plugins. One local storage plugin, which is responsible for OpenPGP and a backend storage plugin which connects the application to a dedicated server. The OpenPGP plugin is stored in the directory "app/-plugin/openpgp/". The file "template.hbs" shows the configuration page inside the application. This page contains information about the version code and the file path of the GPG files. In addition to that it contains a button for manual synchronisation. The file "controller.js" in this directory is responsible for every action related to OpenPGP. It contains functions for changing the file paths of the GPG files, "secring.gpg", "pubring.gpg", and "trustdb.gpg", and the functions for uploading and downloading the GPG files.
The backend storage plugin provides the functionality to communicate with the implemented server. This plugin is stored in the folder "app/plugin/server". The files "template.hbs" and "controller.js" are responsible for the correct configuration of the server. The domain is entered here and is saved to the settings file. The file "server-synchronisation.js" is responsible for the communication. The functions are called by the file "app/helpers/synchronisation-utility.js" and this file belongs to the core module. The OpenPGP plugin calls the functions of the file "app/helpers/synchronisation-utility.js", which then calls the right synchronisation plugin. In this case the server plugin.

Chapter 9

Evaluation

The outcome of the project is an application that takes the key material on one device and transfers it to another one. In the thesis project it is made possible with a combined use of the application with a backend server. It works automated and no user interference is necessary. The Server implementation is an example of how the communication API of the application looks like and how other servers can be adapted to fit in this schema. With this version of implementation the application is still not finished, but the evaluation will refer to the current state of the application. Missing features like S/MIME are not considered. The missing functionality is mentioned in the future work in the end of this thesis. The following describe how the requirements are fulfilled and how good these approaches are.

9.1 Multi Platform Support

The first challenge for the implementation was the requirement for a multi platform support. This problem was solved by using the frameworks Electron and PhoneGap. With these framework a separated application could be created which use the same code base. Therefore no multiple implementations where necessary. The applications fulfil the functional requirements:

- Synchronisation
- Consistency
- Automation
- Encryption
- Configuration

The synchronisation is solved by connecting the application to the backend server. The consistency is created by adding a version code to the data. This version code gets increased for every change that was made. Automation is available because the files that need to be synchronised are observed and when these files are changed the application is informed. The encryption is done by the core module of the application. There the files get symmetric encrypted with AES-256. The configuration is available for each plugin individually.

9.2 Low Maintenance

In addition to the multi platform challenge also the requirements for the maintenance is solved. The whole application is using the same code base with some necessary differences for the different platforms. These differences are caused by Electron and PhoneGap. Each framework enables native access to the device differently. But each version of the application shares the same code where no native access to the device is necessary. So fixing issues must be done once in the most cases.

9.3 Simplicity

The necessary simplicity is possible with the automation of the application. The user does not have to interfere with the application to start a synchronisation. The knowledge of the standard file paths of the GPG files makes it possible that the user does not need any knowledge about his key material. Furthermore the simplicity is also given with the same user interface through out the different versions of the application.

9.4 Extensibility

With this implementation the application can synchronise GPG file with a dedicated backend server. But the code is already prepared for new ways of synchronisation. These different mechanisms just have to be added in the way that the server backend was added. So with this extensibility the application can be used for several additional synchronisation or securing mechanisms.

9.5 Conclusion

To summarise, the application fulfils the given requirements. And with the important factors, multi platform support, low maintenance, simplicity, and extensibility this application shows what it is capable of. The application enables secure email communication in a multi device environment without being restricted to a certain communication tool, like WhiteOut or Protonmail.

Chapter 10

Gained Experiences

During the implementation some critical problems occurred that needed to be solved in order for the application to run properly on all platforms. They belong to the setup and how all frameworks are able to work together.

10.1 EmberJS and PhoneGap

One of the first problems was the combination of EmberJS and PhoneGap. PhoneGap is really sensitive during the start of the application, so when the start takes to long the application crashes. This occurs when the EmberJS application is to big. Because a web application with EmberJS loads big JavaScript files in the beginning the app did not start reliably with PhoneGap. To avoid this problem a empty HTML page was added. This page includes JavaScript that loads the real page after starting the application on a mobile device. This simple trick completely avoids the problem of a crash during the start, that would be caused by a big application.

10.2 EmberJS and Electron

A different problem was between EmberJS and Electron. All functionality that includes the hard drive is done by libraries of NodeJS. In this particular case that framework is called FileSystem. In order for the library to be available through out the application it needs to be imported in a separate JavaScript file before the rest of the application is loaded. In this file every necessary import is happening. After that import the file it self gets imported by EmberJS and the functionality is completely available inside of the application and can be used to read, write, or observe files.

10.3 Electron and PhoneGap

Another problem occurs between PhoneGap and Electron. Both frameworks enable either the mobile or the desktop application. But the actions that are necessary on each device is handled completely different. To avoid crashes inside the application a separation of this calls is necessary. To be able to differentiate the versions a check is made in the beginning of the application and is saved to a global variable. The content of this variable represents the current platform the application is running. With this trick it is easily possible to differentiate the platforms correctly and to avoid crashes.

Chapter 11

Conclusion

The goal of these question was to define the requirements and a suitable design for a secure and extensible synchronisation mechanism that allow secure email communication in a multi device environment.
This question was answered through out the thesis. It showed that OpenPGP and S/MIME are depending on certain information that contain sensitive information, like private and public keys and trust information. These information must be available on all devices in order to enable secure communication with OpenPGP and S/MIME. The thesis showed the necessary requirements for that application and that these requirements contain a multi platform support, consistency, security, automation, simplicity. All these requirements exist so that the application is adapted by the users. These requirements can be fulfilled by the suggested design. This design consists of a storage backend and a client application. The client application is extensible with plugins and is created with a hybrid implementation for the thesis project. Additionally this project includes a server with a REST API that is required for a communication between the server and the application. The hybrid approach of the client implementation is realised with modern technologies and is available on several mobile and desktop operating systems.

11.1 Future Work

With the current state of the application the functionality of synchronisation is given. Also the synchronisation of OpenPGP key material is possible. From this point several more topics can be solved, so that the application gains more functionality and is more reliable in every situation. With the following topics the application can be extend in functionality and integration to other systems.
Implementation of a merging mechanism: With the implementation of a merging mechanism the current problem of merging files can be solved. With that merging

mechanisms OpenPGP keys could be merged. And for other securing mechanisms a merging mechanisms would be necessary as well.
Integration of S/MIME: The implementation of S/MIME is missing. The current state just support OpenPGP as a securing mechanism that can be synchronised. In addition to that S/MIME should be implemented as well. For a S/MIME implementation the access to the different certificate stores will be the main issue. The goal here would be to find an elegant way to get the information for the S/MIME certificate and than synchronise that to different devices.
Integration of cloud storages and P2P: The integration with other synchronisation mechanisms should be done, in order for the application to be accessible for even more users. Like mentioned in the thesis, other synchronisation mechanisms like cloud storages or P2P are also solution for a synchronisation. In order to do that a proper access to the could file API is necessary and for P2P a way to communicate between the applications must be found.
Integration of TUM secure mail: With the integration to the TUM secure mail the problems of key exchanges inside this system can be solved. With TUM as an organisation a backend server is necessary. This theses provides the information for a necessary REST API that the application can communicate with. The task here would be to implement the server backend with the required REST API and connect it with the existing TUM secure mail server.
Integration of mobile applications: With the integration to existing mobile applications for secure communication, the synchronisation mechanism could also fulfil the needs for the mobile operating systems, as the requirements are different here. The sandboxes around the applications do not allow a exchange like it is possible on desktop devices. Even though the synchronisation application makes the necessary information accessible on the device, the integration is required in order to enable the secure communication.

Bibliography

[1] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[2] B. Ramsdell and S. Turner, “Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification,” RFC 5751 (Proposed Standard), Internet Engineering Task Force, Jan. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5751.txt

[3] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, “OpenPGP Message Format,” RFC 4880 (Proposed Standard), Internet Engineering Task Force, Nov. 2007, updated by RFC 5581. [Online]. Available: http://www.ietf.org/rfc/rfc4880.txt

[4] GnuPG, “What’s new in gnupg 2.1,” https://www.gnupg.org/faq/whats-new-in-2.1.html, Accessed Jan 15, 2016.

[5] T. Hase, “Secure pgp key sync - a proposal,” https://blog.whiteout.io/2014/07/07/secure-pgp-key-sync-a-proposal/, July 2014. Accessed Nov 3, 2015.

[6] Whiteout, “Technology,” https://whiteout.io/technology.html, Accessed Jan 4, 2016.

[7] Protonmail, “Sending a message using pgp/pgp (gpg/gpg),” https://protonmail.com/support/knowledge-base/sending-a-message-using-pgppgp/, Accessed Jan 20, 2015.

[8] KeePass, “Feature lust,” http://www.keepass.info/features.html, Accessed Dec 22, 2015.