Secret Sharing Environment for Remote Cryptographic Services

Secure outsourcing of private keys

Description of Bachelor thesis
In this thesis we show a way to minimize the required trust if cryptographic keys are outsourced to a partially trusted third party service. More specifically we analyze an existing architecture for remotely stored private keys and extend that scheme with advanced cryptography such as threshold secret sharing, zero knowledge proofs and multi-signatures.

Degree course:	Engineering and Information Technology
Authors:	Roger Ellenberger, Tobias Flühmann
Tutors:	Prof. Dr. Rolf Haenni, Prof. Gerhard Hassenstein
Expert:	Dr. Igor Metz
Date:	22.01.2019

Abstract

The European Union has standardized an architecture for remote signing services in the context of proposing the future of digital collaboration between member states. Remote signing means that asymmetric cryptographic keys are outsourced to a trust service provider. The aim is to simplify the key management for the end user, but the standard has some major issues concerning the users' control over these cryptographic keys. End users have essentially no control over the private key. We amplified the idea of remote signing with remote decryption. Remote cryptographic services is the generic term for both signing and decryption.

We want to show that it is possible to make the design of these remote cryptographic services more robust. Besides the end user shall get better verifiability about and control over what trust service providers do with their private keys. Moreover, we provide an example that it is possible to combine modern cryptography with an existing solution.

During this work, we created a proof of concept that provides remote cryptographic services for e-mail security. Based on our previous work in which we have analyzed possible improvements, we extended the architecture for these remote signing services to allow shared decryption using a threshold scheme. In addition, a trust service provider creates a cryptographic proof for each decryption it performs. That proof enables the user to verify that the trust service provider has followed the protocol. We have implemented remote signing using a multi-signature scheme where we adapted the idea of thresholds.

To illustrate the optimized architecture we have created a web browser extension and trust service provider environments that are able to communicate with each other. The trust service providers offer remote cryptographic services via API. The design of the browser extension enables it to handle encrypted and signed messages in conformity with the OpenPGP standard.

Acknowledgements

First, we wish to thank both our tutors, Prof. Dr. Rolf Haenni and Prof. Gerhard Hassenstein, for their guidance and ongoing support during this project.

We further thank Dr. Igor Metz for supervising this work as our expert.

Additionally, we thank all the people in the background supporting us creating this work. Namely we wish to thank Martin A. Meier for his extensive proofreading of this report.

Thank you to the OpenPGP community for the outstanding support. Especially, Jon Callas, Heiko Stamer and Derek Atkins for sharing their knowledge regarding ElGamal security in PGP.

Contents

Abstract i

Acknowledgements iii

1. Introduction 1
- 1.1. Background 1
- 1.2. Drawbacks 1
- 1.3. Previous Work 2
- 1.4. Goal of this work 2

2. Technical Background 3
- 2.1. Remote cryptographic services 3
  - 2.1.1. Trustworthy Systems Supporting Server Signing 3
  - 2.1.2. Architecture 3
  - 2.1.3. Shortcomings 4
  - 2.1.4. Technological solutions 5
  - 2.1.5. Variants 6
- 2.2. Scope 6
- 2.3. Trust Assumptions 7
  - 2.3.1. Main assumptions 7
  - 2.3.2. Assumptions as a consequence of the chosen scope 7
- 2.4. Definitions 7
  - 2.4.1. PGP Key 7
  - 2.4.2. Common DSA 8
  - 2.4.3. Multiple DSA 8
  - 2.4.4. Mathematical definitions 8

3. Results 9
- 3.1. System Architecture 9
  - 3.1.1. Client 9
  - 3.1.2. Back-end 12
  - 3.1.3. Trust Service Provider (TSP) 12
  - 3.1.4. Public Key Server (PKS) 13
- 3.2. Application flow 13
  - 3.2.1. User Registration 13
  - 3.2.2. Key Generation 14
  - 3.2.3. Decryption 15
  - 3.2.4. Encryption 17
  - 3.2.5. Signing 17
  - 3.2.6. Verifying 18
- 3.3. OpenPGP 20
  - 3.3.1. PGP public keys 20
  - 3.3.2. PGP messages 20
- 3.4. Cryptography 21
  - 3.4.1. ElGamal encryption scheme 21
  - 3.4.2. Shamir secret sharing 22
  - 3.4.3. Threshold ElGamal 23
  - 3.4.4. Zero Knowledge Proof 24
  - 3.4.5. DSA 25

3.5. Attack scenarios . . . 26
3.5.1. Dishonest TSP - signing . . . 26
3.5.2. Dishonest TSP - decryption . . . 27
3.5.3. Tapping man in the middle . . . 27

4. Discussion **29**
4.1. Future work . . . 30

5. Conclusion **33**

Declaration of authorship **35**

Nomenclature **37**

Glossay **39**

Acronyms **40**

Bibliography **43**

List of figures **45**

List of tables **47**

List of code listings **49**

APPENDICES **51**

A. Project Management **51**
A.1. Agile Method . . . 51
A.2. Documentation . . . 51
A.3. Course of the project . . . 52
A.4. Requirements . . . 55
A.5. Responsibilities . . . 62

B. Evaulation **63**
B.1. Cryptography . . . 63
B.2. email encryption and signature standards . . . 63
B.3. Client technology . . . 64
B.4. Back-end Technology . . . 65

C. Implementation Details **67**
C.1. Add-on . . . 67
C.2. Back-end . . . 68
C.3. OpenPGP . . . 70
C.4. Cryptography . . . 77

D. User Documentation **87**
D.1. Prerequisites . . . 87
D.2. Installation . . . 87
D.3. Add-on setup . . . 87
D.4. Cryptographic operations . . . 91

E. Content of CD-ROM **97**

1. Introduction

These thesis is about dealing with remote cryptographic services. This term means that asymmetric cryptographic operations such as signing or decrypting a message does not happen no longer locally on a user's machine. Consequently, the private key used to perform these operations is not stored on the user's device anymore. This is unusual compared to the classical approach, where users try to protect their private keys as well as possible. Sometimes users even use a smart card from where the key cannot be stolen. With our approach of remote cryptographic services, the user hands over the control of the private key to a third party service. The main goal is to simplify the key management while simultaneously improving the security level. We think it is a fair assumption that a key may be stored more securely when handled by professionals. Combined with strong authentication a remote cryptographic service might be a good solution for many users.

1.1. Background

The blueprint for such a system originates from the European Union. The EU has defined the fundamentals for digital collaboration between member states in the eIDAS regulation. eIDAS stands for electronic IDentification, Authentication and trust Services. A part of the planned ecosystem are remote signing services, called Trustworthy Systems Supporting Server Signing (TW4S). Remote signing is a remote cryptographic service specifically for signing. Such remote signing infrastructures play an important role for the level of trust between EU member's services and their exchanged data. However, the fact that private keys of multiple users and firms are stored in a single place increases the level of risk significantly. With big risks come many compliance checks. The technical base for an audit of such a remote signing service is the standard CEN/TS 419-241 [37, 36] by ETSI.

With a system that is certified by CEN/TS 419-241 a user can create qualified, legally validated signatures. Originally, this used to be exclusively possible for signing keys that were stored on a smart card. Many users might welcome this shift not requiring special hardware. Smart cards still are looked at as a very secure way to store keys. Nevertheless, it is not compatible with all devices. While notebooks getting thinner and lighter, interfaces as smart card slots tend to disappear. The second option of USB smart cards are compatible with most computers but still not with smartphones or tablets. The mobile market trends to gain even more significance in the future.

CEN/TS 419-241 enables the development of applications which bring qualified signatures to a wider range of platforms. A key aspect of ETSI's standard is strong authentication. To create legally valid signatures a user must use two single factor or multi factor authentication. One of these factors should always be a hardware device. However, other than a smart card the authenticator may be integrated in a notebook or smartphone; for instance a TPM or secure element, respectively. A user carries around a legit authenticator implicitly with his device, instead he always needs to have a smart card with him[1]. We suppose that applications can reach a good level of usability following this approach while still provide strong private key protection.

1.2. Drawbacks

The new standard has introduced new problems by trying to simplify the key handling. From our point of view, these issues are not worth a flawless usability. The new problems are actually highly questionable from a user's perspective regarding the trust assumptions. The third party service needs to be fully trusted, since it has all the power over the private key. Being in charge of the private key means that it is technically possible to impersonate the user in the digital world or being able to suppress decisions made by the user. Regulations defined in the standard prohibit this, though. Still, we think the trend is heading in the wrong direction. We prefer technically robust solutions instead of sole regulatory measures.

---

[1] We assume that today a user intuitively takes care of his mobile devices and presume that a user's device needs to be actively managed to guarantee an appropriate level or security.

1.3. Previous Work

In a previous work [7] we have analyzed this standard and criticised that a provider of such a system does not need to add any kind of verifiability for the cryptographic operations. The standard completely suppresses that the user needs to ultimately trust into the service provider since he has absolutely no control over the private key. We have focused our analysis mainly on the use case of remote decryption and introduced two major shortcomings together with possible attack scenarios. Additionally, we proposed how to make such a scheme more resistant. We discuss the results of this analysis in the next chapter.

1.4. Goal of this work

We do not turn down the idea of remote cryptographic services, but we object that technically ETSI's design is kind of the straw man solution and far from ideal. In this work we want to build a proof of concept, which demonstrates that the use of modern cryptography can effectively increase the level of security of a remote cryptographic service. Additionally, we want to present an example that it is possible to combine less known cryptography (in broader use) with existing solutions. For this objective, we take the use case of e-mail security in the first place.

2. Technical Background

In this chapter we provide a technical introduction into the topic. First, we illustrate the design of a remote signing service or a so-called Trustworthy Systems Supporting Server Signing (TW4S). In addition, we introduce the remote decryption service as well along with possible attacks and a short description of the improvements we proposed in our previous work [7]. Finally, we describe the scope of the implemented proof of concept along with some definitions we use throughout this document.

2.1. Remote cryptographic services

2.1.1. Trustworthy Systems Supporting Server Signing

The main principle behind remote signing services as Trustworthy Systems Supporting Server Signing (TW4S) is not too difficult to comprehend. The system consists of a client and a server component. In figure 2.1 we illustrate the process by the example of signing a PDF file. The client creates a distinct and deterministic identification for the PDF file using a cryptographic hash function. This identification is called the hash value, which is a sequence of bits that can be represented for instance as a hexadecimal value. The client then sends this value through a secure channel to the server component where the private key is stored, and the very hash value is signed. The server returns the signature to the client, which inserts it into the PDF file. Anyone in possession of the public key is now able to verify the PDF's signature.

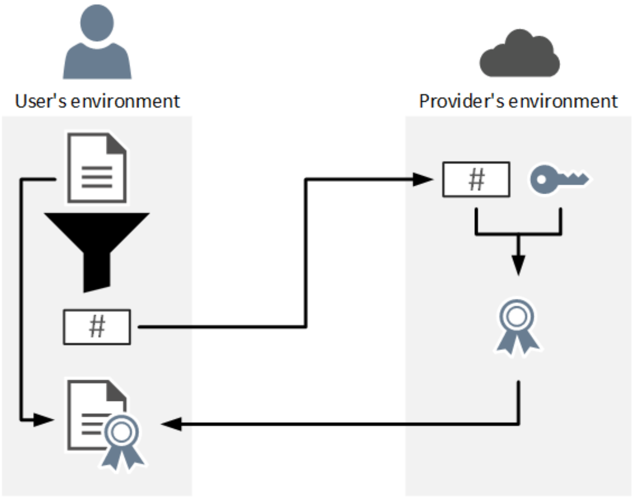


Figure 2.1.: High-level overview of a TW4S

The example is heavily simplified, and it disregards important parts like authentication, specific hardware like HSMs, compliant logging, and key activation. We do not focus on these aspects in this work but ETSI's standard CEN/TS 419-241 [37, 36] explains them in detail.

2.1.2. Architecture

It is possible to adopt the process of remote decryption based on the remote signing as explained in the previous section. In figure 2.2 we use the same example as before, but with an encrypted PDF. Again, the system uses a fairly simplified client-server-architecture. The client accepts an encrypted PDF. It extracts the encrypted symmetric key, opens a secure channel and sends it through that channel to the server. The server decrypts the received data using the private key of the user and sends the result back to the client. The client then can decrypt the whole PDF using a symmetric encryption algorithm.

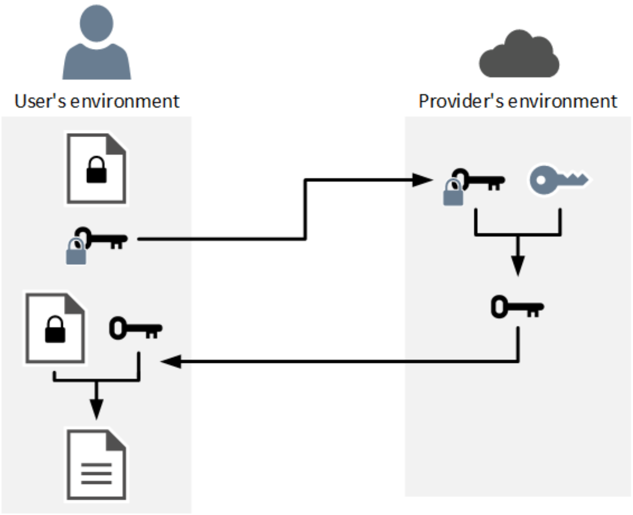


Figure 2.2.: High level overview of a remote decryption service

2.1.3. Shortcomings

Our contribution in [7] included the analysis of the weaknesses of a system for remote decryption. Summarized, there are two kind of problems.

Lack of control over the private key The private key is the weakest point of public key cryptography. A lot of research effort has put into breaking public key cryptography, but the major schemes still seem to withstand the effort to break it. The schemes provide privacy as long as a private key is stored in a safe place. Outsourcing that key does not help to solve that issue. Firstly, the user needs to have ultimate trust in the service. In addition, the risk for the provider raises the more keys he stores in a single place.

Lack of verifiability of received messages A user may not know the content of the encrypted data he sends to the server. The client gets just data returned. The returned plaintext cannot be technically associated with the original ciphertext since the encrypted data contains randomization. Simply re-encrypting the received data would not give any evidence about the correctness of the decryption. Aside from the fact that this approach would be inefficient.

Possible attacks against remote decryption services

These two weaknesses result in potential attacks that are listed below. We use the following exemplification to illustrate the scenarios:

Bob sends an e-mail to Alice. Alice decrypts the e-mail using a remote decryption system provided by a trust service provider (TSP). The TSP performs the decryption and returns the result to Alice's client where she is able to read the content. Mind that Alice sends only the encrypted symmetric key to the TSP.

- **Denial of service** The TSP can send back arbitrary values. This would lead to a failure of the symmetric decryption on Alice's client.

- **Broken privacy** A TSP is technically able to decrypt any of Alice's messages without her knowledge. In a scenario where Alice's e-mail provider and the TSP are closely related or identical, there might be quite an easy possibility to eavesdrop on incoming mails for Alice. These could then be decrypted using the asymmetric on the TSP.

Keep in mind that the use of hybrid encryption does not essentially decrease the risk for such attacks.

Possible attacks against remote singing services

We have not especially analyzed attacks against Remote Signing Services in [7], but the mentioned shortcomings "Lack of control over the private key" and "Lack of verifiability of received messages" apply to remote singing services as well. However, the verifiability of received messages is far less critical. We want to highlight the following possible attacks.

- **Denial of service** Similar to decryption, the TSP could refuse signing a message for the user and thus infantilize him.
- **Impersonation** The TSP is able to sign any messages in Alice's name, since he own's the public key.

2.1.4. Technological solutions

We believe that building a robust system requires that such shortcomings are improved on the technical layer. Trying to fix a technologically weak system without improving it in the core leads to massive compliance requirements. But technically the system itself still does not provide robustness. Therefore we support evolving the system on the core and making it significantly more robust by using modern cryptography, in particular using zero knowledge proofs and secret sharing. We have introduced the use of these technologies in [7] and give a brief summary of the main points below.

Zero Knowledge Proofs Zero knowledge proofs (ZKP) allow the receiver to verify the result the trust service provider returned to the client. It is an efficient method to hedge cryptographic protocols. The use of cryptographic protocols forces all participating parties to follow the predefined rules. For each step, a party needs to deliver a proof that it has performed the very step correctly. A party is actually still able to cheat, but others can detect the misbehaviour and exclude the cheating party from further rounds. Using ZKPs could help the shortcoming "lack of verifiability of received messages". In practice, this means a client can make sure with negligible chance that the received plaintext corresponds to the sent ciphertext.

Secret sharing This technology enables splitting up information into several shares, which then can be distributed among several parties. No party learns anything about the secret if not all parties contribute their share. It is also possible to define a threshold of a minimum number of shares that one needs to reconstruct the secret. A threshold especially improves the availability of the scheme and provides some more protection against a denial of service attack from individuals or small parties (depending on the chosen threshold).

A threshold secret sharing scheme can be evolved to split up a private key of an asymmetric cryptographic scheme. In case of encryption, several parties can perform a partial decryption. None of the parties can learn anything about the secret from the result of the partial decryption. Only the combination of the required number of shares recover the plaintext. This sort of scheme would help reducing the risk that the private key is outsourced and offers a good mitigation against the shortcoming "lack of control over the private key". In a system using threshold encryption, a user may store the key shares on providers of his choice. We believe, this could help to reduce the risk the third party service could use the outsourced key(s) against the user's will.

We discuss the technical details of both technologies in the next chapter.

2.1.5. Variants

The cryptographic primitives ZKP and secret sharing allow us to evolve both the encryption scheme and the signing scheme used in a remote cryptographic service. In this section we want to give you an overview of the available variants.

Decryption

1. **Common encryption** The basic scheme with a single private key. Only a single TSP setup is possible.
2. **Threshold encryption with trusted dealer** The private key for encryption is shared among several parties using threshold secret sharing. A trusted dealer performs the key generation process. A setup with several TSPs is possible.
3. **Threshold encryption with DKG** The same scheme as in 2, but the key generation process happens using distributed key generation process (DKG). The key never exists in compound form, but the key generation process is more elaborate.

The use of ZKPs can be combined with all of the three cases.

Signing

1. **Common signatures** The basic scheme with a single private key. Basically, the scheme ETSI proposed as TW4S with only a single TSP.
2. **Multi-signatures** Several instances of the basic scheme are combined. Several TSPs are involved and each one holds an entire signing key that is not shared or related with the key of other TSPs. Receivers of such messages need to check several signatures.
3. **Threshold signatures with trusted dealer** Identical to variant 2 for decryption but for signing.
4. **Threshold signatures with DKG** Identical to variant 3 for decryption but for signing.

3. and 4. could benefit from the use of ZKPs. Since the plaintext is known, 1 and 2 would not benefit of the use of ZKPs

2.2. Scope

In this project, we implement a proof of concept of a remote cryptographic service infrastructure. We develop an add-on for Mozilla Firefox as a client that communicates with different web services that act as trust service providers. These web services provide cryptographic services via an API.

The very problem for what we provide a solution is the trust into the parties that stores the private key. We focus on demonstrating the cryptographic improvements in regard to trust compared to a single TSP setup in the context of this thesis. For the sake of simplicity we assume certain security relevant points of remote cryptographic services as trivial to achieve.

Therefore, we do not explicitly handle (storage, protection, revocation) the private keys as required in CEN/TS 419-241. We perform secure randomness generation only with best effort. Traceability and account management such as auditing, archiving or role-based access models are not implemented, nor are backup and restore capabilities, maintenance, monitoring, and operation features. Trust in the authenticity of a key is neither a part of this project as well. Communication between clients is not encrypted and authenticated on the application layer since secure channels could be implemented on transport layer. In respect of the implementation, we only evaluate third party components regarding their usability not their security.

The detailed requirements we fulfill are listed in the appendix section A.4.

2.3. Trust Assumptions

We formed our trust assumptions based on the previously defined scope in section 2.2. We first show the most important assumptions which are relevant for the essence of this work. In addition, we present other assumptions we made to simplify the implementation. For these issues we briefly state which currently known technologies are used to solve these issues.

2.3.1. Main assumptions

Based on the following assumptions, we designed our proof of concept.

- **Trust in service providers:** A single trust service provider is not fully trusted.
- **Trust in user's device:** The device of a user is a trusted platform.
- **DDH:** We assume the Decision Diffie-Hellman (DDH) problem [2] is hard. This is relevant since we use discrete logarithm based cryptography in this project.

2.3.2. Assumptions as a consequence of the chosen scope

There are several other assumptions that were made based on the chosen scope of this project:

- **Key management** An adequate key management (storage, protection and revocation) is not necessary.
- **Key properties** Certain properties of keys such as lifetime and randomness are insignificant.
- **Transport security** Implementing transport security is not mandatory.
- **Authenticity of keys** Keys that are stored on the public key repository can be trusted.

2.4. Definitions

2.4.1. PGP Key

There is a common differentiation between keys and certificates in IT security. A bare key is only a sequence of bits (different forms are possible) but has no meta-data associated with it. It is only the key which is used for cryptographic operations. This applies to private and public keys equally. A private key and a public key relate to each other mathematically. However, a bare key has no information which public key that might be.

On the other hand, a certificate is a collection of attributes together with a key and a signature over that data by the owner himself or a third party.

In the context of PGP, the term “PGP key” is frequently used. It does not stand for a bare key but actually a certificate. However, the term “PGP certificate” is barley used in practice [9]. We adopt the common terminology as follows:

- **PGP key:** A certificate in PGP format according [9].
- **PGP public key:** The same as a PGP key but explicitly containing only a public key.
- **PGP private key:** The same as a PGP key but containing both a private and a public key. The private key might be protected by a passphrase.

In case we talk of a bare key we simply call it “private key”, “public key” or “(private/public) key pair”.

2.4.2. Common DSA

We use the term “common DSA” to explicitly distinguish between the ordinary DSA algorithm (as described by NIST in [29]) and Threshold DSA algorithms. This obviously means a common DSA signature has been created using ordinary DSA algorithms and a threshold DSA signature has beencreated using Threshold DSA algorithms. Mind that the verification algorithms might be the same for both algorithms.

2.4.3. Multiple DSA

Multiple DSA describes a multi-signature scheme that is based on common DSA signatures. Other than Threshold DSA, it consists of several common DSA signatures which need to be individually validated. A Threshold DSA signature is just a single signature but created by several cooperating parties.

2.4.4. Mathematical definitions

Mathematical variables used throughout this report are defined in the Nomenclature.

3. Results

ETSI defined the architecture of remote signing services. We have adopted this architecture for our project and extended it to solve the problems mentioned before.

3.1. System Architecture

The key components are the client on one end and the back-end on the other end of each trust service provider (TSP). Additionally, there is a public key server (PKS) for storing OpenPGP public keys. The following diagram (figure 3.1) shows each component with its modules, its interfaces and dependencies, excluding external libraries.

The following sections will describe each component individually.

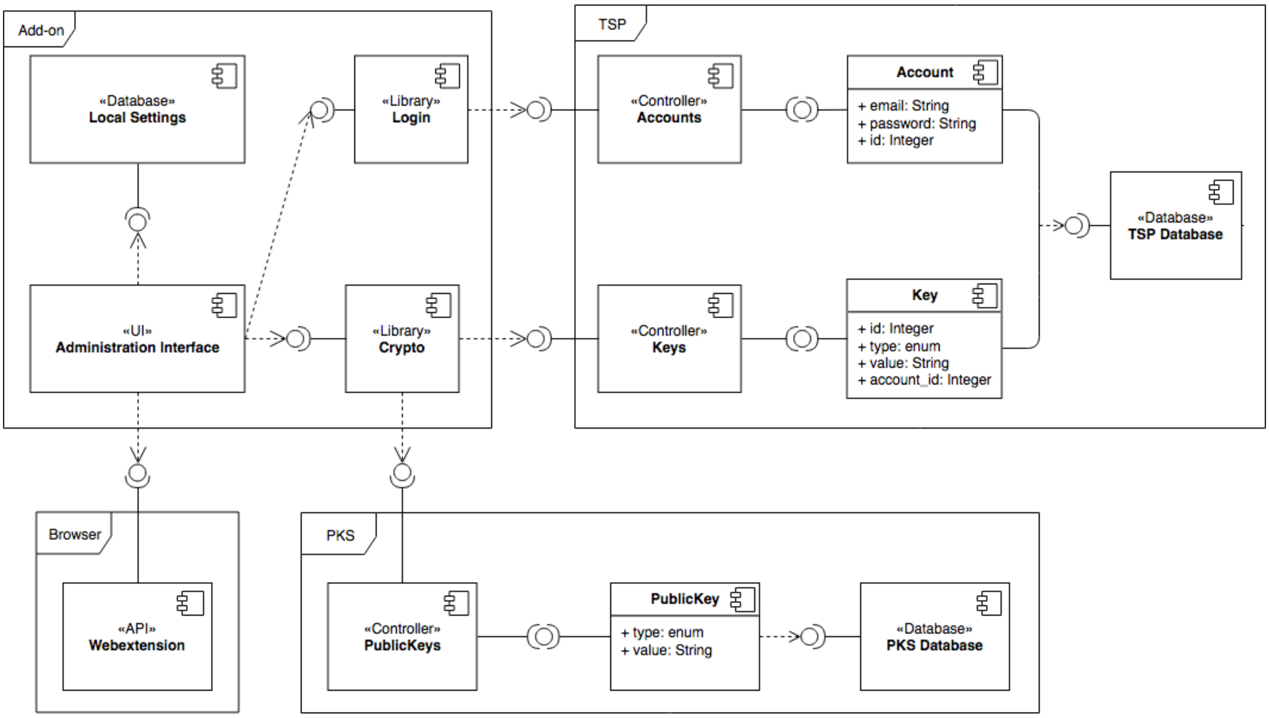

Add-on
«Database»
Local Settings
«Library»
Login
«UI»
Administration Interface
«Library»
Crypto
TSP
«Controller»
Accounts
Account
+ email: String
+ password: String
+ id: Integer
«Database»
TSP Database
«Controller»
Keys
Key
+ id: Integer
+ type: enum
+ value: String
+ account_id: Integer
Browser
«API»
Webextension
PKS
«Controller»
PublicKeys
PublicKey
+ type: enum
+ value: String
«Database»
PKS Database

Figure 3.1.: System Architecture - Component diagram

3.1.1. Client

In our case, the client is a browser add-on for Mozilla Firefox. The considerations leading to this platform decision are documented in section B.3 of the appendices. The add-on is built using the WebExtension standard. This standard allows simpler porting to other browsers like Google Chrome or Microsoft Edge than former add-on standards. WebExtension defines the structure of the add-on and the APIs which browsers have to provide. Not all browsers support all defined API functions. We focussed on the APIs available on Firefox which may lead to some rework of the add-on, in case someone wants to port it to another platform in the future.

Structure

An add-on is loaded using “manifest.json”. This file contains all the references to the other documents and meta data about the add-on such as author and version. We have structured our add-on according to the guidelines provided by Mozilla [24] as illustrated in figure 3.2.

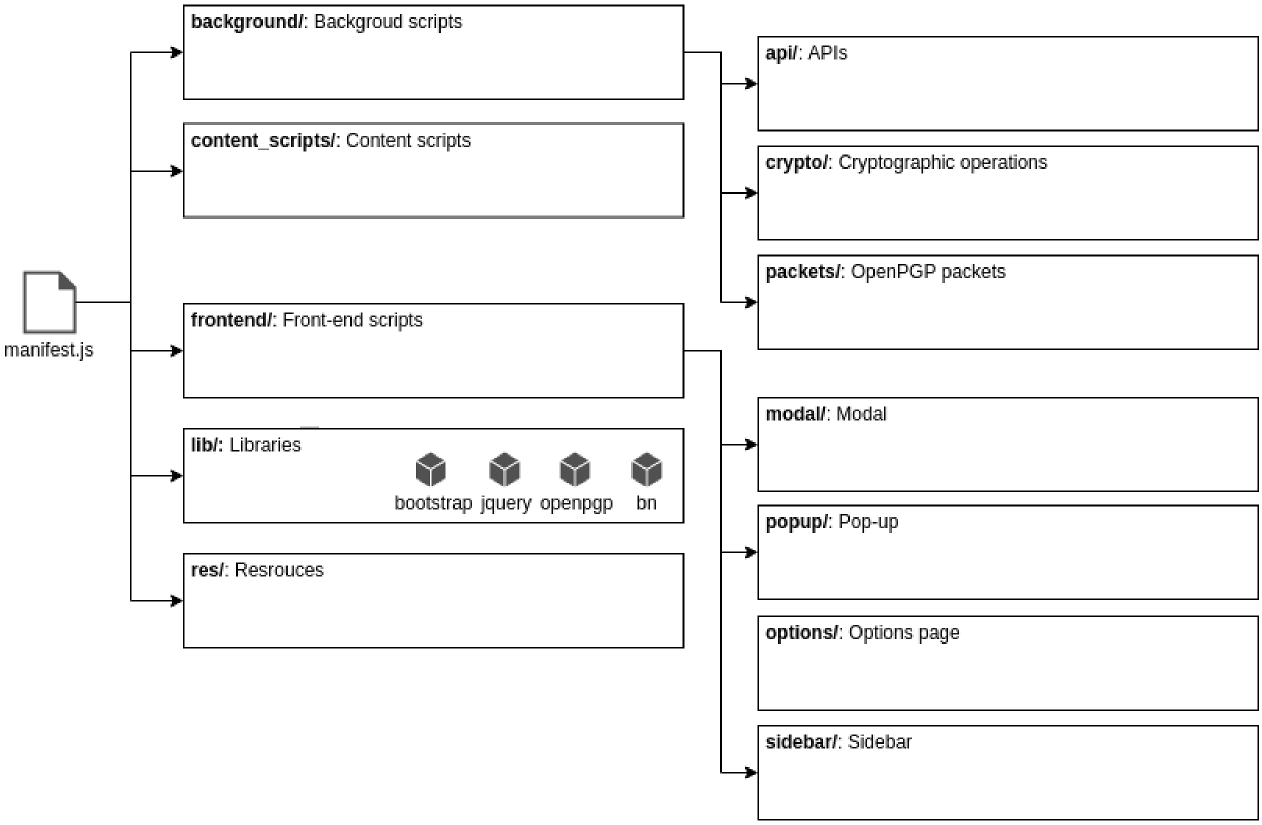

manifest.js
background/: Backgroud scripts
content_scripts/: Content scripts
frontend/: Front-end scripts
lib/: Libraries
bootstrap
jquery
openpgp
bn
res/: Resrouces
api/: APIs
crypto/: Cryptographic operations
packets/: OpenPGP packets
modal/: Modal
popup/: Pop-up
options/: Options page
sidebar/: Sidebar

Figure 3.2.: File structure of addon

Background scripts The browser runs background scripts at load time of the add-on and allows providing functionality for other components. In our case:

- **APIs** Contains abstractions for API communication with the trust service provider (TSP) and public key server (PKS).
- **Cryptographic operations** Code located here provides implementations of cryptographic algorithms.
- **OpenPGP packets** All the classes representing OpenPGP packets are located here.

Content scripts JavaScript placed under content scripts run in the same context as the scripts of the current website (referred to as “page scripts”). We use content scripts to manipulate the DOM of a website.

Front-end Add-ons use front-end scripts to interact with the user through a provided graphical interface. They are built like a common website and can use multiple frameworks or libraries. They cannot interact directly with the document object model (document object model (DOM)) of a website.

- **Modal** Our extension adds the modal as a hidden element to each website the user visits, by appending it to the websites DOM. After the user triggered the use of a cryptographic operation, the modal displays the result of that operation by showing the modal on top of the website’s content. The modal is loaded using the content script. If the modal could not be inserted into the website, the result is displayed using the JavaScript alert box.

- **Pop-up** A pop-up is one of the user interfaces. This component is usually hidden, unless the user clicks on an icon in the browser's menu bar. The pop-up uses HTML documents, CSS style sheets and JavaScript files.
- **Options page** The options page allows the end user to customize the add-on experience. In our case, this is a mandatory component because the functionality we provide is user specific and requires individual configuration. Similar to the pop-up, this is a part of the user front-end and therefore consists of HTML, CSS and JavaScript files.
- **Sidebar** The sidebar is another component of the user interface. It displays details about performed operations and tasks. As the other front-end components, it consists of HTML, CSS and JavaScript files.

Libraries All external code in form of libraries is included in this directory of the add-on. This code may be included in background scripts or used in the front-end.

Resources Static resources are stored in this directory.

Communication between components

Different possibilities for the communication between the components of an add-on exist. The fundamental decisions are briefly stated in section C.1. Figure 3.3 is an abstract illustration of the possible communication between the components.

The add-on makes mostly use of long-lived connections. In WebExtension these connections are called ports. In other cases, we use a feature that allows front-end activity to use a back-end function. This functionality is used to fetch data from the PKS. More can be found in section C.1.1.

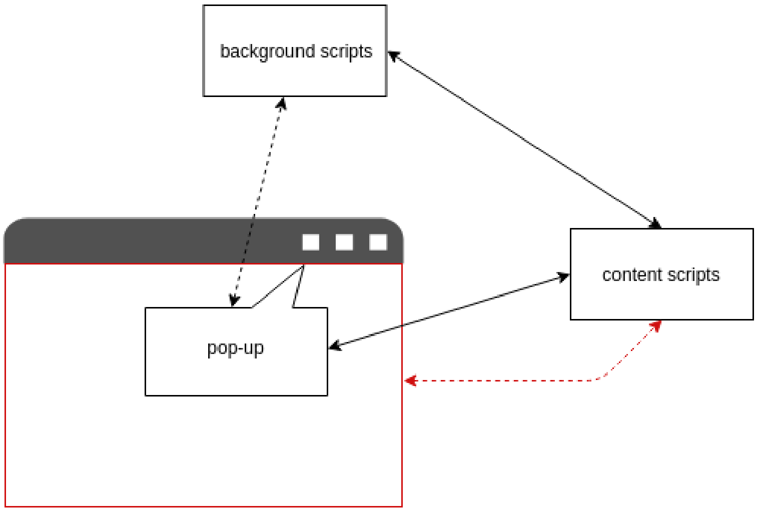


Figure 3.3.: Communication flow between the add-on's components

When the user accesses a website, the content script of our add-on adds the modal to the body of the website. When a user opens the pop-up the first time, the JavaScript within the pop-up makes a call to the background script to get details about what element of the DOM will be manipulated. After that initial call, the message flow goes from the pop-up to the content-script and from the content-script to the background script. The background script performs operations and returns the result to the content script, which then alters the DOM. In our case, the content script puts the received information into the body of the modal and pops it up above the website's content.

Local Storage API

The Firefox WebExtension API allows add-ons to store data. This feature is called local storage. It is based on the Web-Storage API and provides methods to store and retrieve data. Local storage supports listening for changes via Event-listeners. Data is persistent even when the history is cleared or the browser is restarted. The browser deletes stored content only on explicit request or on removal of the add-on from the browser [22]. The Local Storage API allows our add-on to store credentials of the user for each of the TSPs. Additionally, we store the metadata of a key pair, like URLs, PGP public key and partial private key index. This information is represented as JSON in the local storage, but we cast it into JavaScript objects internally. For simplicity reasons some data is stored redundantly.

3.1.2. Back-end

We implemented the back-end components (TSP and PKS) in Ruby[1]. The implementations are self-contained and can be spawned multiple times on different ports. Each instance has its own database. Locally we use a sqlite3 database, but it is easily replaceable with any relational database that our selected ORM Active Record supports. The RESTful API is implemented using the framework grape. "[It provides] a simple DSL to easily develop RESTful APIs" [13]. In order to prevent a user from accessing keys of other users, we implemented policies with the library pundit. The policies are very simple, either it checks if the desired account is the same as the logged in account or it checks if the user is logged in. We have only implemented authentication and authorization, but we have not implemented any accounting features except the standard log provided by the standard Ruby Webrick[2] application server.

3.1.3. Trust Service Provider (TSP)

On the TSP, we provide a small set of endpoints for our add-on to call. The add-on can create an account for the user, add a key, and execute cryptographic operations with the key. Accordingly, the database diagram is very simple as shown in figure 3.4. The API is authenticated using basic-authentication. The API documentation is available on the CD-ROM in chapter E.

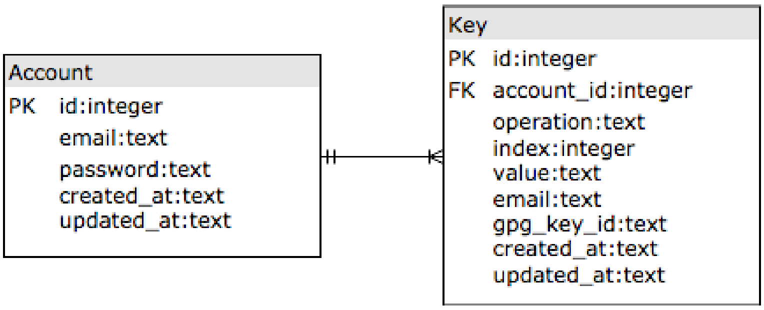


Figure 3.4.: ERD for TSP

Decryption

For the decryption part, we implemented ElGamal partial decryption. The caller of the Application programming interface (API) selects the desired key and the message to decrypt. Since we are using a static group setup for ElGamal, the parameters are statically defined on the TSP. The partial decryption is calculated as described in subsection 3.9. In addition, each TSP adds a proof that it has correctly performed the operation. Subsection 3.12 provides more details about the proof.

---

[1] https://ruby-lang.org (accessed: 23.12.18)

[2] https://ruby-doc.org/stdlib-2.5.1/libdoc/webrick/rdoc/WEBrick.html (accessed: 08.01.18)

Signing

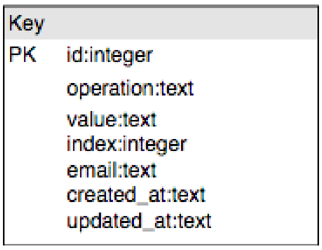
The API for signing a message is similar to the decryption API described above. We have not implemented the functionality for signing by ourselves. We have reused the GNU Privacy Guard[3] implementation for the key generation process and for signing a message. The "clear-signed" message is parsed and just the very signature is returned to the add-on without the part of the armored message.

3.1.4. Public Key Server (PKS)

The key repository is a simple web service to access PGP public keys the user trusts in. Remember that the authentication of public key certificates (in our case PGP public keys) was out of scope of this work. But in order to develop our add-on, we had to either reuse an existing service, or create our own implementation. For simplicity reasons, we decided to create our own key repository.

The PKS is implemented as key value store. The service has a basic API with a single model for the key-email mapping in the database (figure 3.5). We implemented endpoints to search a key by the email , or by key-id. Adding and deleting key endpoints are implemented as well. The API documentation is available on the CD-ROM in chapter E.

This implementation of a PKS without any user-based authentication, offers hardly any privacy. Everyone knows every other participant, but to demonstrate the main aspects of this project, it is good enough.



Key	
PK	id:integer
	operation:text
	value:text
	index:integer
	email:text
	created_at:text
	updated_at:text

Figure 3.5.: ERD for PKS

3.2. Application flow

This section presents the application flow of the different functionalities. For simplicity reasons, the following processes are described using only two TSPs. The described schemes are not limited to a certain number of TSPs.

3.2.1. User Registration

We have implemented a simple onboarding mechanism in our add-on as illustrated in figure 3.6. The user defines an email address and a password. In a second step, the user defines the trust service providers he wants use to store his partial keys. The add-on creates a user account for each TSP the user has configured.

To guarantee that the add-on authenticates on all TSPs using different credentials, we create a separate password for each one derived by the master password. This prevents a TSP from impersonating the user against other TSPs. The password is generated by using the Password-Based Key Derivation Function 2 (PBKDF2) algorithm defined in RFC2898[32]. Since we developed a proof of concept, we derive the password using a static salt that is cryptographically weak. Usually the salt should be fresh and randomly generated.

---

[3] `https://gnupg.org/` (accessed: 11.01.2018)

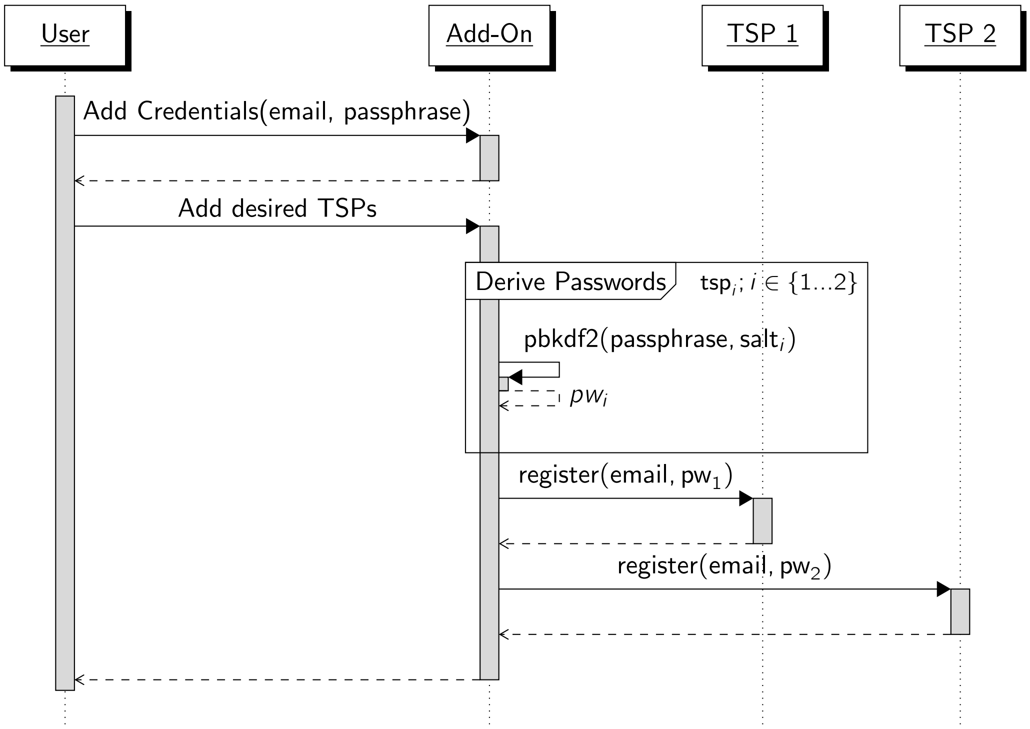

User
Add-On
TSP 1
TSP 2
Add Credentials(email, passphrase)
Add desired TSPs
Derive Passwords
tsp$_i$; $i \in \{1...2\}$
pbkdf2(passphrase, salt$_i$)
pw_i
register(email, pw$_1$)
register(email, pw$_2$)

Figure 3.6.: User Registration Process

3.2.2. Key Generation

The add-on provides a GUI to generate a new key pair for either signing and verifying or encrypting and decrypting.

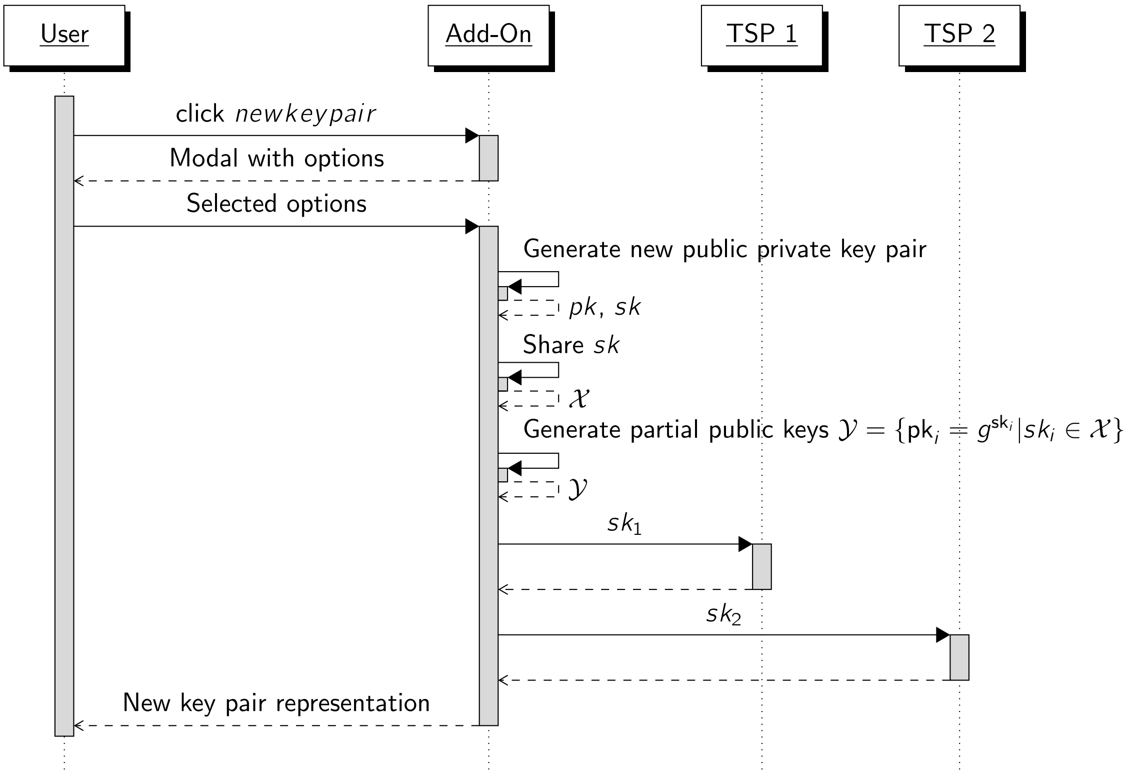

User
Add-On
TSP 1
TSP 2
click $newkeypair$
Modal with options
Selected options
Generate new public private key pair
pk, sk
Share sk
$\mathcal{X}$
Generate partial public keys $\mathcal{Y} = \{\mathrm{pk}_i = g^{\mathrm{sk}_i} | sk_i \in \mathcal{X}\}$
$\mathcal{Y}$
sk_1
sk_2
New key pair representation

Figure 3.7.: Key Generation Process for ElGamal

The user can select the TSPs he wants to collaborate with in the key generation process. Additionally, he can select a threshold that defines the minimum required participating TSPs in the cryptographic operations. The internal process in the add-on for signing and decryption is different. This is due to the decisions at the beginning of the project, how we will implement different cryptographic operations.

We used a typical threshold key sharing scheme with ElGamal for encryption and decryption. The key pair is generated locally in the add-on and the private key is split into multiple pieces as figure 3.7 shows. We generate one single PGP public key certificate for all the key-shares (see section C.3.1 for more details about the PGP public key creation). After splitting the private key we generate a partial public key for each partial private key in order to verify the zero knowledge proof delivered for each partial decryption by the TSP.

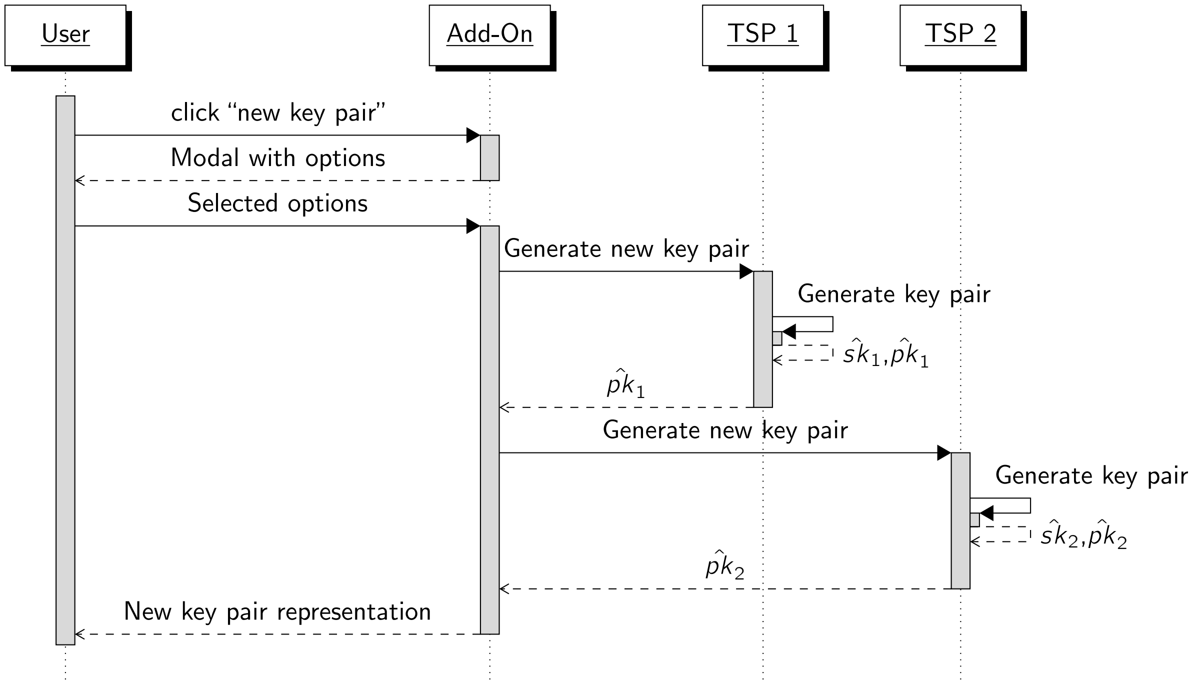


Figure 3.8.: Key Generation Process for DSA

On the other hand, we implemented signing and verifying as a multi-signature scheme (see figure 3.8). Every participating TSP generates a new key pair using GnuPG and returns the PGP public keys $\hat{\mathcal{Y}}$ to the add-on. This has no conceptual reason, but is easier to implement. The PGP public keys are stored in the browsers local-storage (see subsection 3.1.1).

3.2.3. Decryption

The decryption process for the asymmetrically encrypted symmetric session key is distributed over the selected TSP, because we implemented an ElGamal threshold scheme using Shamir's Secret Sharing. The process for the user is straightforward as figure 3.9 shows. The user only has three interactions with the plugin before he gets the plaintext.

First, the user selects the encrypted message. Then he selects the PGP private key representation for the decryption. The PGP private key representation holds the information for each key-share and the corresponding TSP. At the beginning of the process, the add-on parses the selected message and split it into multiple PGP packages (see in C.3.2 for more details). We have focused on the case of plaintext messages without compression in order to simplify the implementation. At this point of the process, we have two packages. The first one is the asymmetrically encrypted symmetric session key and the second one is the encrypted message. The asymmetrically encrypted session key consists of the ElGamal encryption parts c_1 and c_2. The c_2 message part is kept within the add-on, while the c_1 part is sent to the TSPs holding a key-share. Each TSP should calculate a partial decryption d_i using its key-share sk_i (see also equation 3.9).

In order to prove the calculation was correct it has to add a ZKP. The add-on checks the received zero knowledge proofs using the partial public keys that have been generated during the key generation phase (see subsection 3.2.2). If more or equal of the given threshold responses are correct, the partial decryptions d_i for $i \in \{0, ..., n\}$ are combined using Lagrange (see equation 3.9). If not, a message shows up in the sidebar. The user might consider leaving out a potentially dishonest TSP for further cryptographic operations by generating a new key pair, excluding that very TSP. Finally, the outcome of the Lagrange function is combined with c_2 as described in equation 3.3, which results in the decrypted symmetric key. More details about the symmetric key are described in subsection C.3.2 in the appendices.

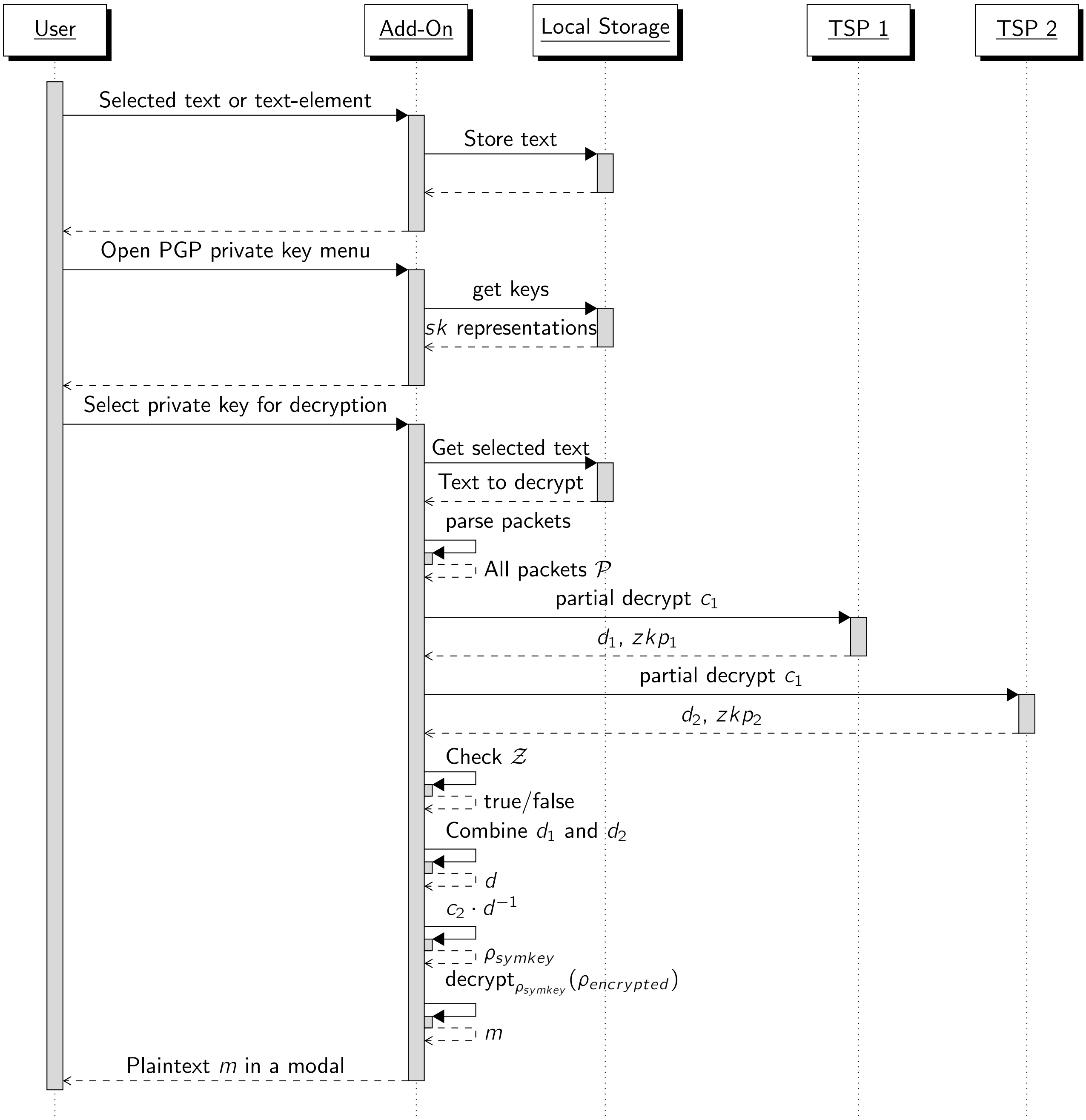

User
Add-On
Local Storage
TSP 1
TSP 2
Selected text or text-element
Store text
Open PGP private key menu
get keys
sk representations
Select private key for decryption
Get selected text
Text to decrypt
parse packets
All packets $\mathcal{P}$
partial decrypt c_1
d_1, zkp_1
partial decrypt c_1
d_2, zkp_2
Check $\mathcal{Z}$
true/false
Combine d_1 and d_2
d
$c_2 \cdot d^{-1}$
ρ_{symkey}
decrypt$_{\rho_{symkey}}(\rho_{encrypted})$
m
Plaintext m in a modal

Figure 3.9.: Decryption Process

3.2.4. Encryption

The encryption process is implemented within the add-on and has only an external interaction to get the PGP public keys, as figure 3.10 describes. The user can select any PGP public key with encryption capabilities that is stored on the PKS (see subsection 3.1.2). The user can select a text on the website and chooses which PGP public key he would like to encrypt with. The add-on downloads this key and uses it to encrypt the selected text using OpenPGP.js. Before the encryption process we use OpenPGP.js again to de-armor the PGP public key, since it is stored in an armored format on the PKS. After encryption has finished, the encrypted text shows up in the modal.

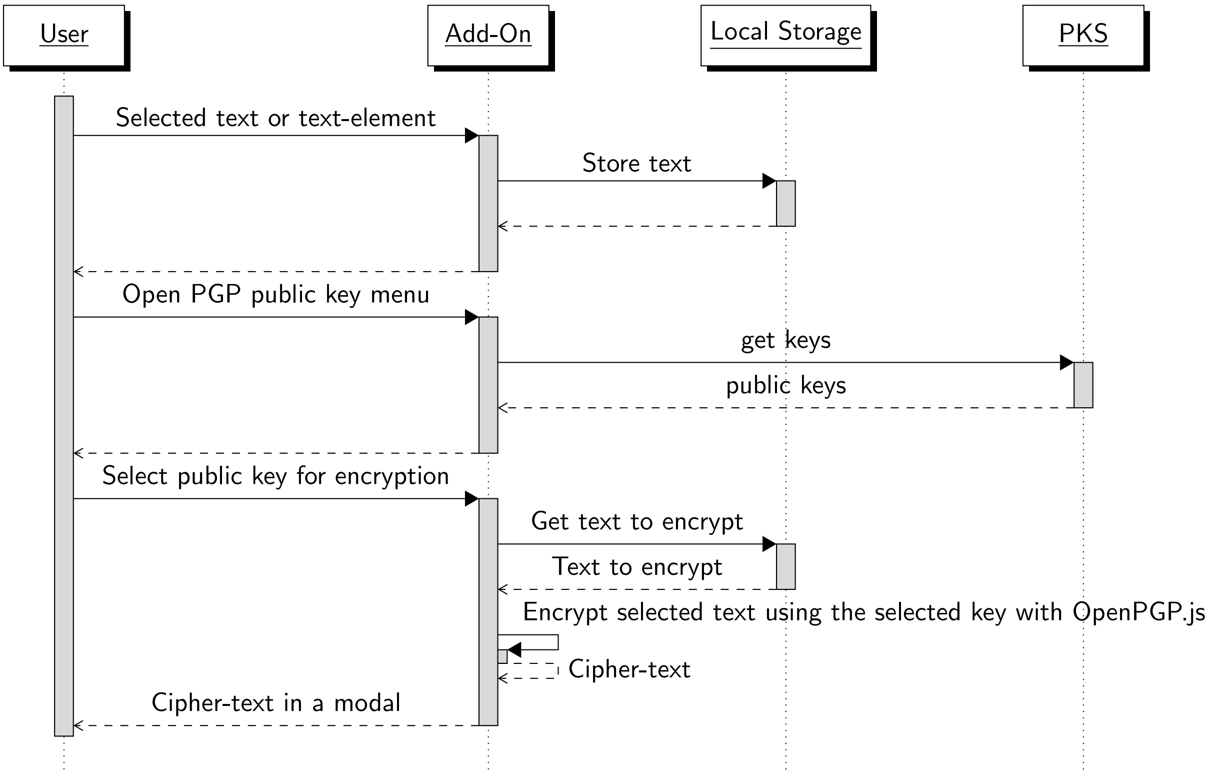

User
Add-On
Local Storage
PKS
Selected text or text-element
Store text
Open PGP public key menu
get keys
public keys
Select public key for encryption
Get text to encrypt
Text to encrypt
Encrypt selected text using the selected key with OpenPGP.js
Cipher-text
Cipher-text in a modal

Figure 3.10.: Encryption Process

3.2.5. Signing

The signing process is based on common DSA and is illustrated in figure 3.11 below. We use multiple common DSA signatures. In this multiple DSA scheme we have adopted the threshold approach, which means only a defined number of signatures have to been hold, that the multi-signature counts to be valid. As described in section 3.2.2 our add-on sends a message to the TSP to create a signing only key during the key generation process. If the user wants to sign a document, he selects the text and chooses the PGP public key that TSPs shall use for signing. The add-on sends an indicator of the PGP public key together with the text to a signing endpoint of the TSP. The response is a so-called "clear signature" for the given text created with common DSA. In the add-on, we compose the text, additional information about the threshold, the participants and all signatures together to an OpenPGP like format.

The goal of the signing format is to stay as compatible as possible with the existing standard. Therefore, the add-on arranges multiple GnuPG "clear signatures" after each other at the end of the message as shown in listing 3.1. The TSP information part includes the information about the participating TSPs and the threshold in a JSON format. The JSON format is Base64 encoded and prepended by a header, which indicates the TSP information. The TSP information itself is part of the message, so it is signed as well. For the 1-out-of-1 case, our scheme is compatible with OpenPGP. In other cases, the validation with a client other than our add-on may fail.

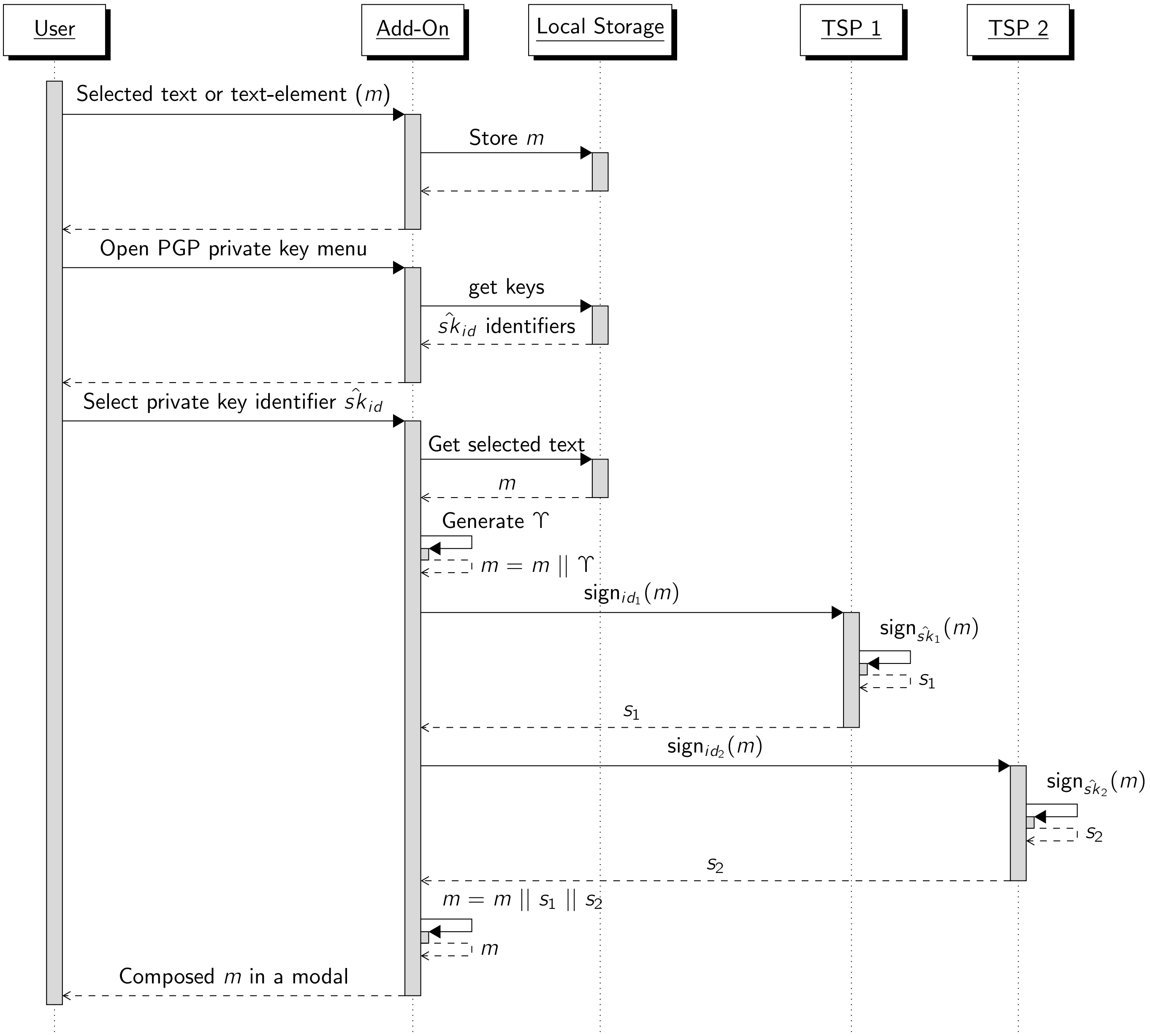

User
Add-On
Local Storage
TSP 1
TSP 2
Selected text or text-element (m)
Store m
Open PGP private key menu
get keys
$\hat{sk}_{id}$ identifiers
Select private key identifier $\hat{sk}_{id}$
Get selected text
m
Generate Υ
$m = m \,||\, \Upsilon$
$\text{sign}_{id_1}(m)$
$\text{sign}_{\hat{sk}_1}(m)$
s_1
s_1
$\text{sign}_{id_2}(m)$
$\text{sign}_{\hat{sk}_2}(m)$
s_2
s_2
$m = m \,||\, s_1 \,||\, s_2$
m
Composed m in a modal

Figure 3.11.: Multiple Common DSA Threshold Signing

3.2.6. Verifying

In the verification process a (multi-) signature is verified. As mentioned in the previous section, the plaintext consists of two parts: The actual message of the user and additional data that describes the involved TSPs and the given threshold. Both parts are signed. The user initiates the verification process by selecting the desired PGP public key(s) as we describe in figure 3.12. In order to check each signature's soundness, the client parses the message and splits it up into the message and several PGP signatures. While the add-on parses the message, it extracts the additional data containing the TSP information and the given threshold. The add-on compounds the message with each of the signatures and verifies it one after the other using the corresponding PGP public key that has been downloaded before from the public key server (PKS). OpenPGP.js provides the verification functionality. If more or the same number of valid signatures are detected compared to the threshold, the multi-signature is valid.

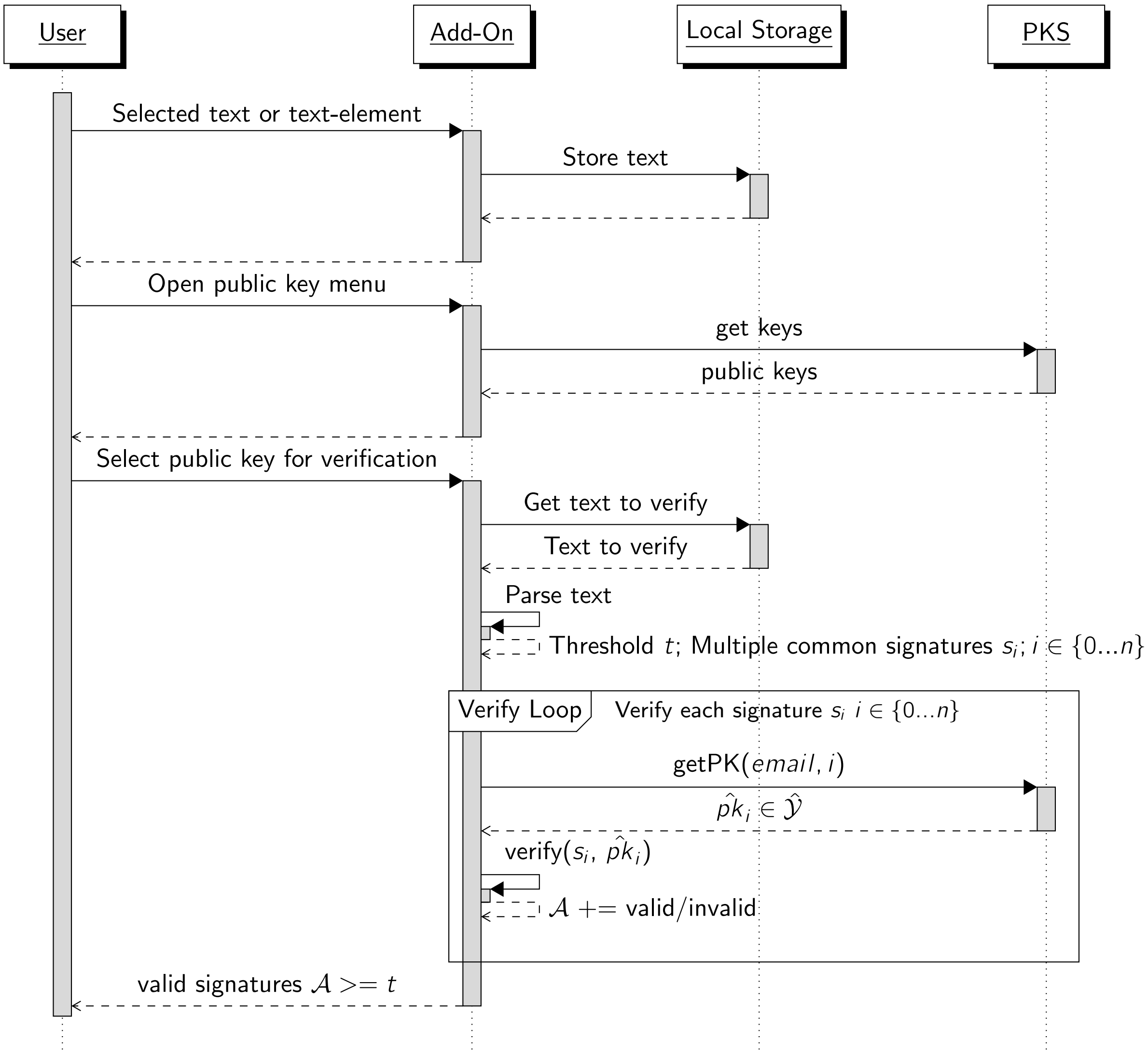

User
Add-On
Local Storage
PKS
Selected text or text-element
Store text
Open public key menu
get keys
public keys
Select public key for verification
Get text to verify
Text to verify
Parse text
Threshold t; Multiple common signatures s_i; $i \in \{0...n\}$
Verify Loop
Verify each signature s_i $i \in \{0...n\}$
getPK($email$, i)
$\hat{pk}_i \in \hat{\mathcal{Y}}$
verify(s_i, $\hat{pk}_i$)
$\mathcal{A}$ += valid/invalid
valid signatures $\mathcal{A}$ >= t

Figure 3.12.: Verification Process

Listing 3.1: Multiple Common DSA Threshold Signature Schema

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA256

Message

- -----BEGIN TSP INFORMATION-----

Base64(JSON({"threshold", "participants", "email", ["index of tsps"]}))

-----BEGIN PGP SIGNATURE-----

Signature 1

-----END PGP SIGNATURE-----
[...]
-----BEGIN PGP SIGNATURE-----

Signature n

-----END PGP SIGNATURE-----
```

3.3. OpenPGP

We briefly present the challenges and design decisions using OpenPGP in our work. OpenPGP is defined in RFC4880 [9]. We attempted to dive into OpenPGP as little as possible, but in the end we dealt with cases where we needed to assemble our own PGP messages on packet level. There are two main reasons for that. Both are related to each other. Firstly, the lack of features in the used library and secondly, we did not want to introduce new dependencies for our client software.

3.3.1. PGP public keys

On the client side, OpenPGP.js provides support for encryption and decryption using ElGamal, as well as singing and verifying using DSA. However, the library lacks a key generation function for both cryptographic algorithms. OpenPGP.js uses JSON Web Algorithms (JWA) in the key generation process, which supports neither ElGamal nor DSA. JWA supports ECDSA though and so does OpenPGP.js, but we use DSA with finite field cryptography, not elliptic curve cryptography. We did not want to extend the existing functionality inside OpenPGP.js nor did we want to use another library (it is even questionable if there had been one). Therefore, we wrote the required data structures to create a PGP public key on our own.

A PGP public key roughly has the following format by convention:

- Public signing key
- User information about key holder
- Self-signature of key
- Public encryption sub key
- Self-signature of sub key

These packets have many fields and sub packets providing details about the key and the signature. We describe these in depth in section C.3.1 in the appendices. Since we decided to use ElGamal and DSA for encryption and signing respectively (see section B.1 in the appendix), the signature key and the encrypting sub key are certainly created using these algorithms. The self-signatures of both are created using the DSA private key that is mathematically related to the public signing key. The process of creating a self-signature is called “certification” of a key.

3.3.2. PGP messages

An OpenPGP message consists of two parts:

- Public key encrypted session key
- Symmetrically encrypted data

We decrypt the first part using Threshold ElGamal as described in section 3.2.3. As soon as the session key is available, the add-on performs the symmetric decryption of the second part. More details about OpenPGP message are described in the appendix section C.3.2.

3.4. Cryptography

In this section we present the details of the cryptography we use in this work.

3.4.1. ElGamal encryption scheme

ElGamal is a well-known encryption scheme by Taher Elgamal. He published his work [6] back in 1985. In the same paper, he proposed the design of the ElGamal signature scheme, which became the basis for the Digital Signature Algorithm (DSA) [39].

We use ElGamal encryption because it is much simpler and more efficient to implement as a threshold scheme compared to RSA. More about this is discussed in the Appendix B.1.

Key generation

In the key generation process a random number in $\mathbb{Z}_p$ is picked. That is the secret key. The public key is generated by calculating a generator by the power of the secret key and reduce the result by the modulus p. When using ElGamal p must be a so-called safe prime. A safe prime is a prime number of the format $p = 2q + 1$ whereas q is prime as well.

$$\begin{aligned} &sk \in_R \{1, ..., q\} \\ &pk := g^{sk} \mod p \\ &sk \in \mathbb{Z}_q, pk \in \mathbb{G}_q \end{aligned} \tag{3.1}$$

g is a generator of the group $\mathbb{G}_q$. Any value that is picked as the power of g and then is reduced modulo p maps into this group. $\mathbb{G}_q$ is of prime order q.

Encryption

A plaintext $m \in \mathbb{G}_q$ is encrypted using a public key pk and a random value $r \in \mathbb{Z}_q$. The resulting ciphertext consists of two parts called c_1 and c_2.

$$\begin{aligned} r \in_R \{1, ..., q\} \\ enc_{pk}(m, r) : \mathbb{G}_q \times \mathbb{G}_q \times \mathbb{Z}_q \rightarrow \mathbb{G}_q \times \mathbb{G}_q \\ enc_{pk}(m, r) := (g^r, m \cdot pk^r) = c = (c_1, c_2) \end{aligned} \tag{3.2}$$

Decryption

The decryption function merges the three spaces $\mathbb{Z}_q, \mathbb{G}_q$ and $\mathbb{G}_q$ back into the message space $\mathbb{G}_q$

$$\begin{aligned} dec_{sk}(c) :&\mathbb{Z}_q \times \mathbb{G}_q \times \mathbb{G}_q \rightarrow \mathbb{G}_q \\ dec_{sk}(c) : &= c_2 \cdot c_1^{-sk} \\ &= (m \cdot pk^r) \cdot (g^r)^{-sk} \\ &= (m \cdot (g^{sk})^r) \cdot (g^r)^{-sk} \\ &= (m \cdot g^{r \cdot sk} \cdot g^{r \cdot -sk} \\ &= m \end{aligned} \tag{3.3}$$

Mapping of plaintext into message space

According to current knowledge, it is required to map the actual plaintext message into the message space to achieve IND-CPA security for ElGamal. Both equations 3.2 and 3.3 show that m is in $\mathbb{G}_q$.

However, OpenPGP does not consider that mapping. Instead, it combines ElGamal with PKCS #1 (Version 1.5) EME encoding. This is a padding scheme including some randomized bytes. There are no references under what assumptions this scheme is secure. We got a reply by the former CTO of PGP Corporation, Jon Callas and others on our post on the mailing list. The reasons are mainly historical. Details about the investigation can be found in the appendix C.4.2.

3.4.2. Shamir secret sharing

Before we introduce Threshold ElGamal we look at the basics. Shamir secret sharing [34] is an efficient way to share a message m among n people, whereas a subgroup of t people suffice to recover the message.

Pick polynom

Shamir's threshold secret sharing scheme is based on a polynomial P of degree $t-1$ satisfying $P(x) = m$. To pick such a polynomial $t-1$ coefficients $a_1, ..., a_{t-1}$ are randomly chosen. The polynomial is then calculated as shown in the equation below.

$$\begin{aligned} &\forall a_i \in_R \{1, ..., p\} \quad \wedge \quad i \in \{1, ..., t-1\} \\ &P(x) := a_0 + \sum_{i=1}^{t-1} a_i \cdot x^i \quad \mid a_0 = m \end{aligned} \tag{3.4}$$

Create shares

A share δ_i is a tuple of the form (i, m_i), where i is the input value into $P(x)$. The outcome of this is $P(i) = m_i$. Hence the sharing space is $\mathcal{S} = ([1, n] \times \mathbb{Z}_p)^n$.

$$\begin{aligned} \forall i \in \{1...n\} :& \\ m_i &:= P(i) \\ \delta_i &:= (i, m_i) \\ \\ share(m) &: \mathbb{Z}_p \to \mathcal{S} \\ &:= (\delta_1, ..., \delta_n) \\ &:= ((1, m_1), ..., (n, m_n)) \end{aligned} \tag{3.5}$$

Recover the message

Only t out of n shares are needed to recover the secret. Still it is also possible to use more than the minimal threshold to recover m. We denote the shares which m is recovered from as $\mathcal{S}' := \{\delta_{j_1}, ..., \delta_{j_t}\}$. Mind that $\mathcal{S}' \subseteq \mathcal{S}$. This case shows the minimum needed shares. Hence, $\mathcal{S}'$ is actually a subset $\mathcal{S}$ here.

$$\begin{aligned} recover(\delta_{j_1}, ..., \delta_{j_t}) &: \mathcal{S}' \to \mathbb{Z}_p \\ &: = P(0) = \sum_{i=1}^{t} m_{j_i} \prod_{1 \le l \le t \wedge l \ne i} \frac{j_l}{j_l - j_i} = m \end{aligned} \tag{3.6}$$

3.4.3. Threshold ElGamal

Combining both techniques, we have just introduced allows us to build a threshold encryption scheme based on ElGamal. The fact that ElGamal is a DL based system brings some useful properties for this application. Threshold ElGamal can be derived from both techniques ElGamal encryption scheme and Shamir secret sharing. Additional sources we have used are [4] by Boneh and Shoup as well as [1] by Groneberg.

Key generation

In general, there are two possibilities creating a shared key pair. A trusted dealer approach or a distributed key generation process. The user has to choose a threshold t and a maximum number of participants n for his shared key pair at the time of key generation process in both cases.

We choose a trusted dealer approach. This approach presumes that the dealer can be fully trusted. Otherwise, the system may be compromised from the beginning. In our case the user himself is the trusted dealer and it is in his own interest that the key generation process is done correctly. Therefore, a trusted dealer creation is justifiable. A distributed key generation process still brings the main advantage that the private key is never stored compositely anywhere during the whole lifetime of the key. This might lead to even better privacy of the private key but since suitable algorithms are more complex, we resign for the time being.

Threshold key generation process First a regular key generation process is performed.

$$\begin{aligned} &sk \in_R \{1, ..., q\} \\ &pk := g^{sk} \mod p \\ &sk \in \mathbb{Z}_q, pk \in \mathbb{G}_q \end{aligned} \tag{3.7}$$

Then the private key is shared using a $t-1$ degree polynomial. The sharing works as described in equation 3.5 with the difference that we share sk instead of m. Note that the message to be shared is in $\mathbb{Z}_q$. In equation 3.5 we used $\mathbb{Z}_p$. Since both p and q are prime there is no difference in the algorithm itself, but note that the sharing space $\mathcal{S} = ([1, n] \times \mathbb{Z}_q)^n$ is different as well.

$$\begin{aligned} share(sk) :\ & \mathbb{Z}_q \to \mathcal{S} \\ &:= (\delta_1, ..., \delta_n) \\ &:= ((i, sk_i), ..., (i, sk_i)) \end{aligned} \tag{3.8}$$

Encryption

The encryption process does not differ from the regular ElGamal. This is one of the big advantages of this scheme and in particular, one of the main reasons we have decided to use it. Any person who wants to send an encrypted message to a user who uses a threshold encryption scheme does not even notice that he does and can encrypt the message as usual. We do not need to do any modifications on the public key for this reason.

Decryption

The decryption on the other hand is different. There are two steps involved. First, each party performs a partial decryption of c_1 into $d_1, ..., d_t$. Then the party who received the message recovers d from the shares. Only t out of n shares are required to recover it. We denote the picked shares as $j_i \in \{1, ..., n\}$, for $i \in 1, ..., t$. This means we pick t indices of the n possible indices.

$$\begin{aligned} partialDec_{sk_{j_i}}(c_1) &: \mathbb{Z}_q \times \mathbb{G}_q \to \mathbb{G}_q \\ &:= c_1^{sk_{j_i}} \mod p \\ &:= d_{j_i} \end{aligned} \tag{3.9}$$

Afterwards it continues with the regular decryption process of ElGamal as already shown. We enrich the partial decryptions d_{j_i} with the index before passing into the recover function: $((j_1, d_{j_1}), ..., (j_t, d_{j_t}))$.

$$\begin{aligned} d &:= recover((j_1, d_{j_1}), ..., (j_t, d_{j_t})) \\ m &:= c_2 \cdot d^{-1} \end{aligned} \tag{3.10}$$

However, the recover function differs a little compared to the one shown above for Shamir's Secret Sharing. As equation 3.11 shows, we do not use a sum but a product for recovering the shares.

$$\begin{aligned} recover((j_1, d_{j_1}), ..., (j_t, d_{j_t})) &: ([1, n] \times \mathbb{G}_q)^t \to \mathbb{G}_q \\ &: = P(0) \\ &= \prod_{i=1}^{t} d_{j_i}^{\lambda_{j_i}} = c_1^{sk} = d \quad \mid \text{where} \quad \lambda_{j_i} := \prod_{\substack{l=1 \\ l \neq i}}^{t} \frac{j_l}{j_l - j_i} \in \mathbb{Z}_q \end{aligned} \tag{3.11}$$

3.4.4. Zero Knowledge Proof

So far the used scheme is able to recover from rogue trust service providers, if at least as much participants as the threshold t act correctly. Nevertheless, the scheme is unable to detect if a TSP is trying to manipulate the result of a decryption. We needed to try several different combinations to recover from the attack if a decryption fails (we need a combination of partial decryption where the cheating TSP or TSPs are not included) and need to try all combinations to find out which TSP(s) has/have violated the protocol. This would lead to time and resource intensive computations.

A much better approach is to use cryptographic proofs to ensure all participants follow the protocol. Commonly used in cryptographic protocols are zero knowledge proof (ZKP). A verifier can check with negligible probability that the proofer delivered a sound response. We use ZKPs to check if a TSP delivered a correct partial decryption without any need to reveal the secret key share.

A client needs a possibility to ensure a TSP$_i$ has used his key share sk_i to perform the partial decryption $d_i = c_1^{sk_i}$. This key share is the same as in $pk_i = g^{sk_i}$ which is the share of the public key. We use a zero knowledge proof that ensures equality between discrete logarithms to achieve that. The proof for equality between discrete logarithms has been introduced in the Chaum-Pedersen protocol in 1992 [3]. Formally we define the non-interactive zero knowledge proof as follows:

$$\hat{t}_1, \hat{t}_2, \hat{c}, \hat{s} := NIZKP\{(sk_i) : pk_i = g^{sk_i} \wedge d_i = c_1^{sk_i}\} \tag{3.12}$$

To minimize communication between the client and the TSP we use a non-interactive proof. The protocol of the proof is shown in figure 3.13. The used $\mathcal{H}$ in the figure is a cryptographic hash function, which has a bit length of $\ell_{\mathcal{H}}$. Remember from the sections 3.4.1 and 3.4.3 that $pk_i, c_1, d_i \in \mathbb{G}_q$ and $sk_i \in \mathbb{Z}_q$. For the sake of completeness mind that $\hat{t}_1, \hat{t}_2 \in \mathbb{G}_q$, $w, \hat{s} \in \mathbb{Z}_q$ and $\hat{c} \in \{0,1\}^{\ell_{\mathcal{H}}}$.

Client (Verifier)		**TSP$_i$ (Prover)**
knows pk_i, c_1		knows sk_i, $pk_i = g^{sk_i}$, d_i
	$\xrightarrow{c_1}$	
		compute $d_i = c_1^{sk_i} \mod p$
	$\xleftarrow{d_i}$	
		pick $w \in_R \mathbb{Z}_q$ compute $\hat{t}_1 = g^w \mod p$ compute $\hat{t}_2 = c_1^w \mod p$ compute $\hat{c} = \mathcal{H}(pk_i, d_i, \hat{t}_1, \hat{t}_2)$ compute $\hat{s} = w + \hat{c} \cdot sk_i \mod q$
	$\xleftarrow{\hat{t}_1,\ \hat{t}_2,\ \hat{c},\ \hat{s}}$	
check $\hat{c} \overset{?}{=} \mathcal{H}(pk_i, d_i, \hat{t}_1, \hat{t}_2)$ check $g^{\hat{s}} \mod p \overset{?}{=} \hat{t}_1 \cdot pk_i^{\hat{c}} \mod p$ check $c_1^{\hat{s}} \mod p \overset{?}{=} \hat{t}_2 \cdot d_i^{\hat{c}} \mod p$		

Figure 3.13.: Protocol flow of non-interactive ZKP

3.4.5. DSA

We had to implement our own solution for DSA signatures adapted by OpenPGP.js to sign a generated ElGamal public key. Since we defined the authenticity of PGP public keys as out of scope (see 2.2), we decided to use a fixed group we are working with. The static group parameters (p, q and g) are defined in C.4.1. In the key generation process, only the private key $\hat{sk}$ is randomly chosen. The public key $\hat{pk}$ is based on the secret key and inherits its randomness. NIST defines the Parameters for DSA in chapter 4.1 of FIPS 186-4 [29]

$$
\begin{aligned}
&\hat{sk} \in_R \{1...q-1\} \\
&\hat{pk} := g^{\hat{sk}} \quad mod(p)
\end{aligned}
\tag{3.13}
$$

For each signature a randomly generated parameter k is required (FIPS 186-4 Chapter 4.5 [29]).

$$
k \in_R \{1...q-1\} \tag{3.14}
$$

The signature generation for a message m is specified in chapter 4.6 of the Digital Signature Standard [29]:

$$
\begin{aligned}
&\hat{r} := (g^k \quad mod(p)) \quad mod(q) \\
&z := \text{the leftmost } min(N, \ell_h) \text{ bits of } \mathcal{H}(m) \\
&u := (k^{-1}(z + \hat{sk} \cdot \hat{r})) \quad mod(q)
\end{aligned}
\tag{3.15}
$$

Finally, the signature s consists of the tuple (r, u).

We do not describe the verification algorithm for DSA because we did not implement it by ourselves. We have reused the OpenPGP.js implementations for the verification use-cases. We implemented signing according NIST's specification [29] and used some code from other sources as described with in-line comments.

3.5. Attack scenarios

In chapter 2 we had a look at few attacks against remote cryptographic services with a single TSP. In this section we have a closer look to attacks and protocol flaws we discovered during the implementation of our environment using multiple TSPs.

3.5.1. Dishonest TSP - signing

Due to a design flaw during development of this scheme, it can be bypassed. The key generation process is outsourced to the trust service provider as described in 3.2.5. Moreover, our "additional TSP Information" in the message can easily be tampered with a dishonest TSP. In our scheme, every TSP uses a common DSA signature key. Every key and with that every TSP itself is able to generate a valid signature in a 1-out-of-n scheme. Therefore, a dishonest TSP could execute a kind of downgrade attack and reduce a t-out-of-n scheme to a 1-out-of-n scheme. The TSP would probably honestly sign messages sent by the client, but it could copy the message and counterfeit the additional TSP information, sign the new message and the outcome would be a validly signed message.

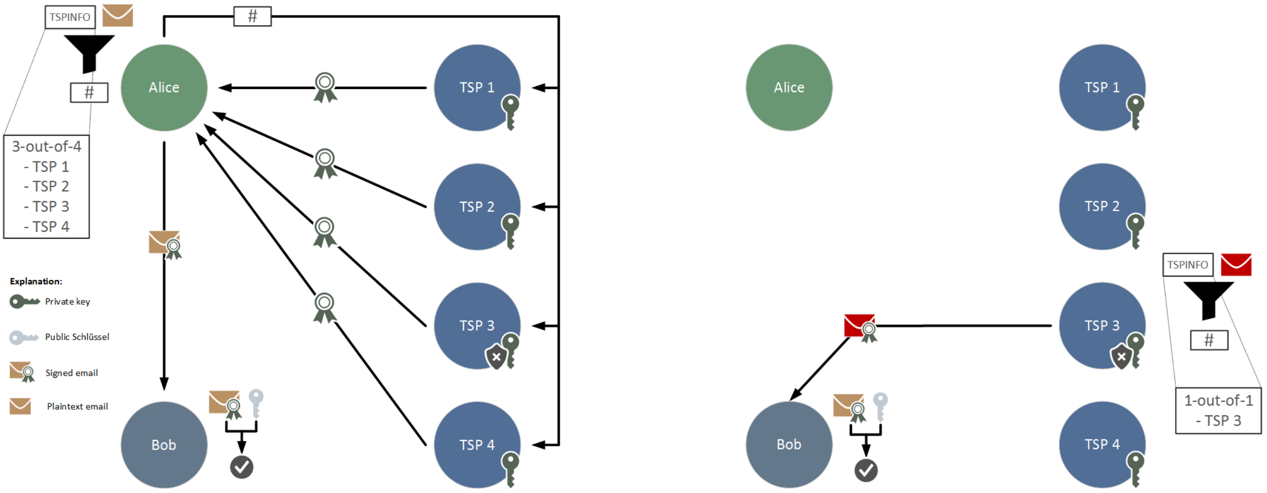


Figure 3.14.: Example of attack against signing implementation with TSP 3 acting dishonest

The flaw was detected after implementation, but we have figured out several solutions how to solve this issue.

Convention

We could prohibit clients to accept messages with a threshold smaller than two by convention. With this rule, the problem could be solved easily. But the problem would still exist technically. An implementation not considering this rule would still be vulnerable against this attack.

TSP information in PGP public key

Removing the additional TSP information (3.2.5) from the message part and adding it to the PGP public key(s) itself would solve the problem as well. The threshold information would be hard-coded into the PGP public key and signed using the PGP private key. In this way the PGP public key would include the information and tampering the information would be impossible. The client could check the correctness of the PGP public key after the key generation process by checking an example signature issued with the PGP private key. However, authentication of such keys becomes even more important. In the first place the scheme can be improved using this solution. Unfortunately, we could not implement this due to lack of time.

Threshold signing using secret sharing

From our point of view, the most elegant way to solve the problem would be to use a threshold scheme similar to decryption. The private key would be split into parts which are distributed over the selected TSPs. The public key would look like a common verification key and would not hold any meta data about threshold or TSPs. Every TSP would contribute to a single signature. The threshold would be implicitly clear to the client. The TSP would not have to know any additional information and could not perform a valid signature on behalf of the user on its own. A scheme like that has already been proposed and implemented in the scientific paper: "Threshold-optimal DSA/ECDSA signatures and an application to Bitcoin wallet security" [10].

3.5.2. Dishonest TSP - decryption

The chosen systems architecture allows detecting if a TSP responded with a wrong decryption. Even in a setup with a single TSP, since the PGP private key is generated on the client. The fact that the client generates the PGP private key helps, because the client can initially generate the PGP public key by his own. With the additional implementation of non-interactive zero knowledge proofs, the client detects a wrong decryption without knowing the plaintext or the private key. Knowing the partial public key $pk_i = g^{sk_i}$ is enough to check if the decryption was performed with sk_i as described in section 3.4.4. If the ZKP would not be implemented, composing multiple partial decryptions with wrong partial decryptions would fail. Without having ZKP implemented, it would be impossible to detect the wrong partial decryption without iterating all possibilities.

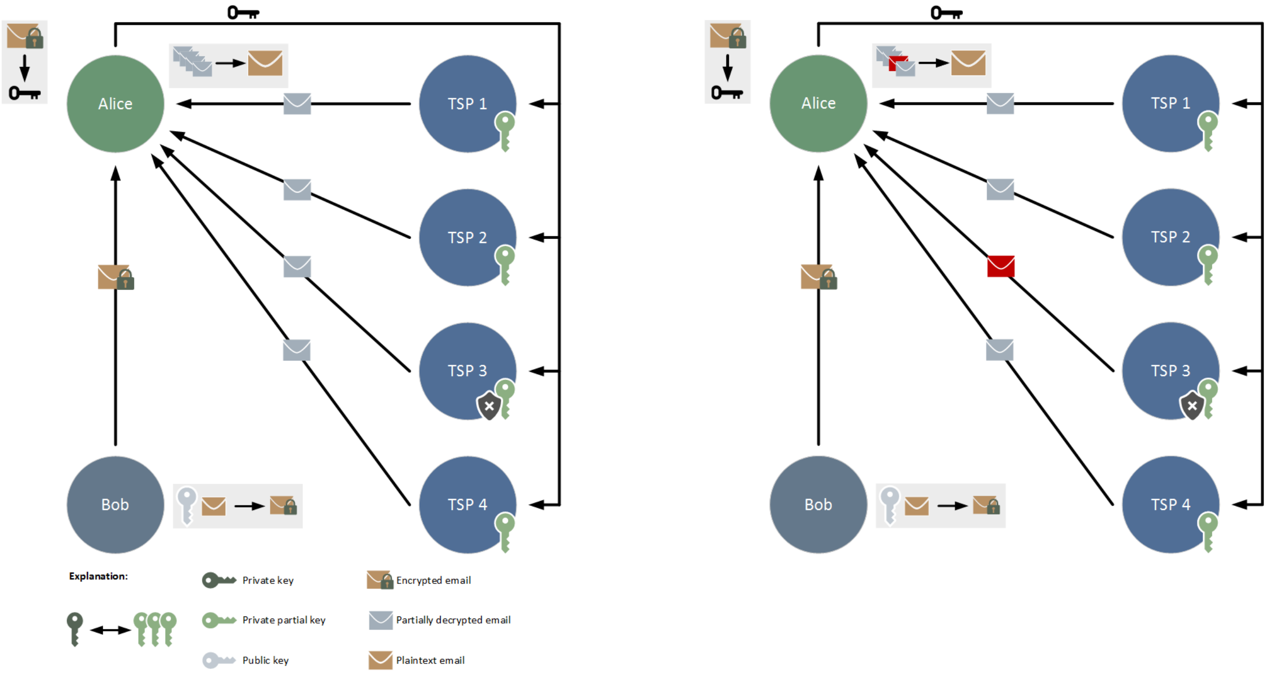


Figure 3.15.: Example of unsuccessful attack against decryption implementation with TSP 3 acting dishonest

Figure 3.15 illustrates the system is even able to recover from an attack unless the threshold is bigger than the number of valid decryptions.

3.5.3. Tapping man in the middle

Today's https connections are not fully protected from being monitored. Furthermore, eavesdropping also known as TLS interception is an inbuilt feature of TLS up to version 1.2 and in the new enterprise TLS version recently published by the ETSI. To protect the user from being monitored, we planned to implement end to end authenticated encrypted communication between the client and each TSP in addition to TLS. The final goal would have been to have end to end encryption between the cryptographic hardware and the client. Due to lack of time, this feature, defined as optional in the requirements in appendix A.4 as `FR.104.3.1`, could not be implemented.

4. Discussion

Our work started with an analysis of a system defined by ETSI in CEN/TS419241-1 and CEN/TS419241-2 in the previous project [7]. We appreciate the effort that is being made to simplify key management for end users. The effect of ETSI's approach of outsourcing keys is quite convenient to use. An end user does not need to have his cryptographic keys stored on every device he might use. Instead he uses strong authentication to access this keys and benefits of a professional key handling by the chosen trust service provider (TSP). On the other hand, we have identified that such a system has several shortcomings. It lacks two major things: Privacy of the encrypted content and control over the private key. Some TSPs claim their systems to be more secure than a consumer's device, which might be correct in some cases. The main issue is the verifiability. A user who decides to outsource his private key to a TSP has to believe and trust that the key is stored in a protected environment and there is an activation process only he can use. He has no possibility to prove that, though. CEN/TS419241 solves these issues using a defined process and auditing mechanisms.

We implemented two different approaches in our thesis. For remote decryption we have made use of Threshold ElGamal, which is basically ElGamal combined with Shamir's Secret Sharing. For verifiability we make use of zero knowledge proof (ZKP)s. On the other hand, we have implemented remote signing using several common Digital Signature Algorithm (DSA) signature combined in a custom scheme. This resulted in a multi-signature scheme that follows a threshold approach. This implementation is significantly simpler, but it has some drawbacks as already discussed in 3.4.5. There are mitigations against these drawbacks. Nevertheless, it is still unclear if the system has other flaws. It is a good example that combining cryptographic primitives in a protocol is not a simple task. Compared to remote signing our decryption implementation is way more robust, since we have used algorithms that were made specifically for such tasks.

In section 2.1.3 of the chapter Technical Background, we summarized the shortcoming for remote cryptographic services in a single TSP scenario. We pick up on these threats and show how they developed using our improved environment.

Lack of control over the private key Lack of control over the private-key could be improved significantly. If several TSPs are involved, a single TSP is no longer able to control the private key by itself. To control the private key several TSPs would need to collaborate. Specifically, as much TSPs as the minimal number of participants (the threshold) was set at key generation. The higher the threshold, the less we need to trust in a single instance.

Lack of verifiability of received messages Lack of verifiability of received messages could be solved using a zero knowledge proof (ZKP) in addition to the decryption. If the ZKP is not correct, the decryption is not accepted by the add-on. If this happens often, a TSP might be considered being suspicious. This lack only relates to decryption, because a signature can be verified easily using the public-key. If one would use a threshold signing scheme, thinking about reusing the concept of ZKP would definitely make sense from our point of view.

Attacks Looking at the attacks, we could heavily improve the resistance of remote cryptographic services.

- **Denial of service** A TSP that denies the service is not a problem for our schemes, if the threshold is chosen properly. For instance, if a 3-out-of-5 threshold has been chosen, two TSPs can regularly send back no or wrong data. The same applies to lost partial keys which was not our intention. Threshold schemes improve the availability of the private-key upon to a certain point. Additionally, zero knowledge proofs help identifying wrong answers, which are not further used in the combining step of the partial decryptions.
- **Broken privacy** In a single TSP scenario, the TSP is technically able to sign and decrypt any message. This might be worse in a corporate environment. The privacy of a person in a single TSP environment can be broken for signing and decryption, if for instance both, the administrator of the email server and the administrator of the TSP work together. Using the proposed threshold schemes with external TSPs and

a proper threshold of participants the privacy cannot be broken trivially anymore. Of course, a threshold scheme does help for individual users as well, since they can choose several TSPs which are not related to each other or even competitors.

- **Impersonation** As described in the shortcomings (2.1.3), in a single TSP environment, a provider can easily forge a message and send it on behalf of the impersonated person. Our scheme helps here, since several TSPs need to provide signatures. As a downside, only our add-on can easily verify the signature. This attack is dedicated to signing.

During our work we have not analyzed what a reasonable number of participants or a reasonable threshold of participants might be. In fact, we have not limited any scenarios that are mathematically possible with Shamir's Secret Sharing. Our scheme still allows a single TSP environment using the 1-out-of-1 case. It allows the 1-out-of-n case as well, even if the problems might become even worse then: More than one TSP would be able to decrypt a cipher-text or sign a message on behalf of the user. Therefore, it makes sense to use a minimum threshold of two participants whereas the number of participants can be equal or bigger depending of the use case. The schemes we have used are not limited to a maximum number of participating TSPs. However, there is a computationally feasible limit on today's consumer systems or at least a limit about what is appropriate for the user to wait for. Especially when it comes to smartphones or any devices with less CPU power like e.g. smart-watches or IOT devices, the waiting period could increase to execute remote cryptographic operation even if it is just RESTful API calls. Additionally, our implementation triggers each TSP sequentially. This limitation is given by design, since it's a Firefox add-on (JavaScript WebExtension) which does not support multi-threading. Technically we could stop triggering the TSPs as soon as the threshold is reached, but our system is built for demonstration purpose, therefore we trigger and show the results returned by all of them.

It could be argued that our implementation requires a customized client application. This is definitely correct, a client must either implement the cryptography we've used by itself or extended with an add-on. But existing solutions according to CEN/TS419241 do have the same requirement, their implementations need a customized client as well.

Our proof of concept solves the drawbacks mentioned before using several technical approaches. Several cryptographic schemes combined as shown in chapter 3 can significantly increase users control over his keys. Distributing the trust over several TSPs reduces the level of trust required for each TSP. Furthermore, it is possible to revoke trust from a suspicious or untrustworthy TSP if required. Malicious TSPs would have to collaborate to recover an encrypted message or to sign a forged document. If the threshold is set properly, this becomes very unlikely.

From our perspective, a technical solution is preferred over a process-based approach when dealing with such significant problems. A process-based solution requires a lot of compliance overhead and is usually very expensive. It might help avoiding the problems, but it will not solve it for good. With the technical approach comes another advantage. The TSP holds less risk if he only manages a partial key for his customers. Firms that act as a trust service provider usually have an extensive risk management and need appropriate reserves. This is one of the aspects that make these services so expensive. Being in charge of only a part of a private key minimizes risk and could lead to a cheaper service. Partial keys do not need the same protection level as composed keys need. Additionally, maybe more trust service providers would be able to fulfill the requirements to store partial keys compared to the requirements for composed private-keys. This could then lead to a broader offering on the market.

4.1. Future work

The product and the idea have potential for enhancement in various areas. The system is not production ready and several optimizations for better user experience are therefore possible. Authentication and performance could be improved as well. Moreover, it would make sense to support other platforms than a browser. If the concept should go on the market, it would be essential to have a broad client support, also for mobile platforms.

From the technical point of view, one of the most important things would be to implement a threshold scheme and zero knowledge proofs for signing as well. This would show in the first place that a Trustworthy Systems Supporting Server Signing can be effectively improved. Another point would be to implement a fully distributed key generation process, which would release trust in the client. We defined distributed key generation as optional in our requirements and did not implement it due to lack of time.

In addition to OpenPGP it might be reasonable to add support for S/MIME as well. S/MIME is more common for email messaging between businesses. The support of cryptographic algorithms is no as extensive as with OpenPGP, but it does have options for DL based cryptography in the standard. It supports Diffie-Hellman and DSA, as well as elliptic curve cryptography.

Defining a standard for threshold schemes with secret sharing using proofs demonstrates the correctness of the work might be the ultimate goal. First of all, the mindset of the industry and the sensitivity of the end-user for privacy has to change, otherwise there is no incentive to continue development in this direction.

Generally, research in the area of threshold schemes is still actively going on. Especially since cryptocurrencies are getting more popular, threshold signing with distributed private-keys could be a way to prevent theft of private-keys for example [10]. We assume there will be further improvements of these schemes in the future. In practice still only a few applications make use of such algorithms, although the basic algorithms using trusted dealer key generation have been around for many years. We would appreciate to see more solutions in the future using use distributed cryptography.

5. Conclusion

Our thesis aimed creating a proof of concept of a remote cryptographic service in a distributed environment. The system focuses on the distributed cryptography but is still as compatible as possible to the ETSI standard. We created a solution for remote decryption and remote signing, but the main focus lies on decryption. The implementation of decryption and signing was done with different approaches.

For decryption we implemented Threshold ElGamal using Shamir's Secret Sharing with additional zero knowledge proofs. We can detect, and to a certain level even recover, from falsified or corrupt messages sent by rogue TSPs. Our system significantly improves the privacy and the control over the private key for the user. For signing we implemented a multi-signature scheme. Information about the trust service providers and the threshold is signed in combination with the message by several TSPs. A common DSA key pair is generated on each TSP. The signatures are created using regular GNU Privacy Guard and appended to the message. The message can only be considered as authentic if a certain number of signatures appended are valid. For this scheme we already discovered some drawbacks. Using a threshold signature scheme seems the best solution to fix these drawbacks and would help the scheme not being trivially tricked anymore. Due to time constraints, we could not implement this solution during our project. But for us it showed once again that it is still the safest option to use a specific scheme that already fulfills the requirements instead of creating an own scheme. Hence, we recommend using Threshold DSA instead of our multi-signature scheme.

The implementation using OpenPGP is a success story from our point of view. In the beginning we have chosen PGP as encryption format to implement the threshold schemes. We were not aware of how deep we needed to understand the different packets in order to implement the key generation and the message parsing part for the decryption. We had to implement several parts we did not expect in the beginning, because they did not already exist in a way, we could have reused them. The "deep dive" into the OpenPGP RFC was very interesting and we discovered several things we would not have found if we would have reused existing parts. A good example is the ElGamal implementation of OpenPGP that uses PKCS #1 EME encoding instead of the today's state-of-the-art message space mapping into $\mathbb{G}_q$ as described in the appendix C.4.2. The additional implementation deepened our knowledge about OpenPGP but prevented an improved implementation of DSA. On the other hand, we might have needed the same knowledge for the implementation of threshold DSA using Shamir's Secret Sharing as well. Therefore, we are happy with the outcome ofthe project.

Declaration of primary authorship

We hereby confirm that we have written this Bachelor thesis independently and without using other sources and resources than those specified in the bibliography. All text passages which were not written by us are marked as quotations and provided with the exact indication of its origin.

Place, Date:	Berne, 22.01.2019	
Last Names, First Names:	Ellenberger, Roger	Flühmann, Tobias
Signatures:	....	....

Nomenclature

i, j, l	Element enumerators
$\hat{pk}$	Composed public key for verification
$\hat{\mathcal{Y}}$	Set of composed public keys $\hat{pk}$
pk	Composed public key for encryption
pk_i	Partial public key, calculated with the related sk_i
$\mathcal{Y}$	Set of partial public keys pk_i
$\hat{sk}$	Composed public key for signing
sk	Composed private key for decryption
sk_i	Share of private key for decryption
$\mathcal{X}$	Set of partial private keys for decryption sk_i
n	Integer of maximum participating parties
N	Bitlength of q
L	Bitlength of p
t	Integer of the threshold participating parties
k	fresh, randomly picked parameter for a DSA signature $\in_R \{1, ..., q-1\}$
z	Leftmost bits of $mathcalH(m)forasignature$
$\hat{r}$	Lefthand side of a DSA signature
u	Righthand side of a DSA signature
s, s_i	DSA signature (r,u) (calculated using $\hat{sk_i}$)
Υ	Additional TSP information specified in 3.1
$\mathcal{A}$	Set of validated signatures $\in \mathbb{B}$
$\mathbb{B}$	Boolean set
m	Message
m_i	Message share, shared using Shamir's Secret Sharing
$\mathcal{P}$	Parsed packets of an OpenPGP message
ρ	Packet of an OpenPGP message $\rho \in \mathcal{P}$
δ_i	Share in Shamir's Secret Sharing, tuple of $(i,\ m_i)$
λ	Function, part of Shamir's Secret Sharing
$\mathcal{S}$	Sharing space in Shamir's Secret Sharing
$\mathcal{S}'$	Subset of $\mathcal{S}$
r	randomized integer from the denoted set
d_i	Partial decryption for ElGamal using sk_i
d	Set of all partial decryptions d_i

c_1	lefthand side of an ElGamal encrypted message
c_2	righthand side of an ElGamal encrypted message
p	Safe prime whereas $q = (p - 1)/2$
q	Prime number
zkp	zero knowledge proof parameters
$\mathcal{Z}$	Set of zkp parameters
$\hat{t}_{1,2}$	Commitment $\in \mathbb{G}_q$
$\hat{s}$	Response to $\hat{c} \in \mathbb{Z}_q$
$\hat{c}$	Challenge $\in \mathbb{Z}_q$
w	Randomly picked element $\in \mathbb{Z}_q$
g	generator of Group $\mathbb{G}_q$
$\hat{g}$	generator of Subgroup of order q in the multiplicative group of GF(p)
$\mathbb{G}_q$	Multiplicative subgroup of integers modulo p
$\mathbb{Z}_q$	Field of integers modulo q
$\mathbb{Z}_p^*$	Multiplicative group of integers modulo p
$\in_R$	Randomly chosen element of the following set
$\mathcal{H}$	Cryptographic hash function
ℓ_h	Bitlength of the cryptographic hash function $\mathcal{H}$

Glossary

armor Armoring is a process for putting metadata around radix64 encoded data, described in rfc4880 section-6.2. 13

CEN/TS419241 Trustworthy Systems Supporting Server Signing. The main objective of this standard is to define requirements and recommendations for a networked signing server which may manage signing keys used by natural or legal persons for the creation of digital signatures. [37, 36]. 29, 30, 39

CEN/TS419241-1 First part of CEN/TS419241, Security Requirements for Trustworthy Systems Supporting Server Signing;. 29

CEN/TS419241-2 Second part of CEN/TS419241, Protection profile for QSCD for Server Signing. 29

Content Security Possible (CSP) Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks. [26]. 67

cryptographic hash function Is a deterministic one way function that fulfills particular propetires required for cryptographic use. Well known algorithms are SHA2 and SHA3. 3, 24, 38

DDH Stands for Decisional Diffie-Hellman and is an assumption for computing complexity [2]. 7

de-armor De-armoring is the opposite of armoring and describes the process to extract data from armored data. It is described in rfc4880 section-6.2. 17

DL DL stand for the computation problem of the discrete logarithm. It is believed to be hard. DL is used in various asymmetric cryptographic schemes. 23, 31, 63

DSL Stands for domain specific language and is a "language that's targeted to a particular kind of problem, rather than a general purpose language that's aimed at any kind of software problem“ [18]. 12, 68

ElGamal ElGamal is an encryption and signature scheme based on the discrete logarithm [6]. iii, 12, 14, 15, 20–25, 29, 33, 37, 38, 45, 52, 63, 67, 70, 77, 79, 80, 83

ETSI Stands for the European Telecommunications Standards Institute which is an European standardization institute. 1–3, 6, 9, 27, 29, 33

HSM HSM stands for hardware security module. A system for secure management of cryptographic keys. 3

IND-CPA Stands for Indistinguishability under chosen-plaintext attack. It is a formally defined game between a challenger and an adversary used to proof semantic security under chosen-plaintext attack. [38]. 79

OpenPGP.js Javascript library implementing the OpenPGP Standard. 17, 18, 20, 25, 67, 70, 75, 76

ORM Object-relational mapping tool. Translates a database relation into an object oriented programming language object. 12, 69

PKCS #1 PKCS #1 is the cryptographic standard for the RSA algorithm. Latest version is 2.2, but 1.5 is still used in some software, e.g. OpenPGP [16, 19]. 22, 33, 76, 79

RESTful Representational State Transfer (REST) provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. [8]. 12, 30, 65, 67, 68

RFC Stands for request for comment and are publications which are supported by the Internet Society ISOC. 33, 63

RSA An asymmetric encryption and signature scheme based on the problem of factorization of the product two large prime numbers. It is standardized as PKCS #1. 21, 79

S/MIME Stands for Secure/Multipurpose Internet Mail Extensions and is a standard for encrypting or digitally sign e-mail messages [30]. 31, 63, 64

Scrum A framework for agile software development or similar team driven work. 51

secure channel A communication channel which is authenticated and encrypted. 3, 4

Shamir's Secret Sharing Secret Sharing Scheme developed by Adi Shamir using polynomial interpolation [34]. 15, 24, 29, 30, 33, 37, 83

smart card A smart card is a device to store cryptographic keys. Keys on a smart card cannot be extracted. Operations such as decryption or signing happen on the device itself.. 1

Test-Driven-Development Test-Driven Development (TDD) is a technique for building software that guides software development by writing tests. [17]. 52, 68

TPM A TPM is a trusted plattform module. An security chip on a motherboard similar to a smart card. 1

UNIX timestamp A 32 bit singed integer counting the number of seconds after 1970-01-01 00:00:00. 72, 73, 76

web of trust A decentralized trust model introduced with PGP as an alternative for centralized trust models such as the one of a public key infrastructure. 63

WebExtension "A cross-browser system for developing extensions" [28]. 9, 11, 12, 30, 64, 67

Acronyms

API application programming interface. 6, 9, 10, 12, 13, 30, 64, 65, 67, 68, 70

DH Diffie-Hellman. 31, 63
DOM document object model. 10, 11, 64
DSA Digital Signature Algorithm. 17, 20, 21, 25, 26, 29, 31, 33, 37, 63, 67, 70, 79
DSS Digital Signature Standard. 25

ECC elliptic curve cryptography. 20, 31
eIDAS electronic IDentification, Authentication and trust Services. 1

FFC finite field cryptography. 20

GnuPG GNU Privacy Guard. 13, 15, 17, 33, 60, 61

IOT Internet of Things. 30

JWA JSON Web Algorithms. 20, 70

PBKDF2 Password-Based Key Derivation Function 2. 13
PGP OpenPGP. iii, 9, 10, 12, 13, 15–18, 20, 25–27, 31, 33, 37, 60, 61, 63, 64, 70–72, 74, 75, 79, 80, 90–95
PKI public key infrastructure. 63
PKS public key server. 9–13, 17, 18, 52, 61, 67, 68, 88, 90, 97

TLS Transport Layer Security. 27
TSP trust service provider. i, 4–6, 9, 10, 12–18, 24, 26, 27, 29, 30, 33, 37, 52, 55, 60, 61, 68, 70, 88, 89, 97
TW4S Trustworthy Systems Supporting Server Signing. 1, 3, 6, 30

ZKP zero knowledge proof. 5, 6, 15, 16, 24, 25, 27, 29, 30, 33, 38, 45

Bibliography

[1] Björn Groneberg. (2013) Threshold Cryptography. https://www.dcl.hpi.uni-potsdam.de/teaching/cloudsec/presentations/threshold-cryptography.pdf. Accessed: 28.10.2018.

[2] D. Boneh, "The decision diffie-hellman problem," in *Algorithmic Number Theory*, J. P. Buhler, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 48–63.

[3] D. Chaum and T. P. Pedersen, "Wallet databases with observers," in *Advances in Cryptology — CRYPTO' 92*, E. F. Brickell, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 89–105.

[4] V. S. Dan Boneh, "A graduate course in applied cryptography," https://crypto.stanford.edu/ dabo/cryptobook/, September 2017, accessed: 14.10.2018.

[5] Derek Atkins, Heiko Stamer, Jon Callas, Watson Ladd, Roger Ellenberger. (2018, December) IEFT Mail Archive - Re: [openpgp] IND-CPA security of OpenPGP's ElGamal implementation. https://mailarchive.ietf.org/arch/msg/openpgp/UJXk68Fu2Kx8m-PYzNzOySVNlP8, https://mailarchive.ietf.org/arch/msg/openpgp/R2NxvBrbtlbEwROdOa2k13XeklQ, https://mailarchive.ietf.org/arch/msg/openpgp/bS0-C2G3AEan5ElghqMxWwVSREs, https://mailarchive.ietf.org/arch/msg/openpgp/Pfa4UUZCtMDpT38CuCGp10JcnJk. Accessed: 10.12.2018.

[6] T. Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, Jul 1985.

[7] R. Ellenberger and T. Flühmann, "Remote secure decryption service," Bern Universitiy for Applied Sciences, Tech. Rep., February 2018.

[8] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, UNIVERSITY OF CALIFORNIA, IRVINE, 2000, accessed: 28.12.2018.

[9] H. Finney, L. Donnerhacke, J. Callas, R. L. Thayer, and D. Shaw, "OpenPGP Message Format," RFC 4880, Nov. 2007. [Online]. Available: https://rfc-editor.org/rfc/rfc4880.txt

[10] R. Gennaro, S. Goldfeder, and A. Narayanan, "Threshold-optimal dsa/ecdsa signatures and an application to bitcoin wallet security," Cryptology ePrint Archive, Report 2016/013, 2016, https://eprint.iacr.org/2016/013.

[11] Google LLC. Message Passing. https://developer.chrome.com/extensions/messaging. Accessed: 20.10.2018.

[12] ——. Overview. https://developer.chrome.com/extensions/overview. Accessed: 20.10.2018.

[13] Grape Community. Grape. https://github.com/ruby-grape/grape. Accessed: 23.12.2018.

[14] Hal Finney, Kimmo Mäkeläinen. (2005, May) IEFT Mail Archive - Re: Problems with calculating signatures over keys. https://mailarchive.ietf.org/arch/msg/openpgp/JUbNte4iyK_26pMEbDu0VKSgO0Y. Accessed: 11.11.2018.

[15] Heiko Stamer. (2017, October) Distributed key generation and threshold cryptography for OpenPGP. https://www.nongnu.org/libtmcg/dg81_slides.pdf. Accessed: 10.12.2018.

[16] B. Kaliski, "PKCS #1: RSA Encryption Version 1.5," RFC 2313, Mar. 1998. [Online]. Available: https://rfc-editor.org/rfc/rfc2313.txt

[17] Martin Fowler. (2005, March) Test Driven Development (TDD). https://www.martinfowler.com/bliki/TestDrivenDevelopment.html. Accessed: 01.01.2019.

[18] ——. (2008, May) Domain Specific Language. https://www.martinfowler.com/bliki/DomainSpecificLanguage.html. Accessed: 23.12.2018.

[19] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2," RFC 8017, Nov. 2016. [Online]. Available: https://rfc-editor.org/rfc/rfc8017.txt

[20] Mozilla. (2017, December) Porting a Google Chrome extension. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Porting_a_Google_Chrome_extension. Accessed: 30.09.2018.

[21] ——. (2017, November) runtime.getBackgroundPage(. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/runtime/getBackgroundPage. Accessed: 02.01.2019.

[22] ——. (2017, August) storage.local. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/storage/local. Accessed: 23.12.2018.

[23] ——. (2018, August) Add-ons. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons. Accessed: 20.09.2018.

[24] ——. (2018, October) Anatomy of an extension. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Anatomy_of_a_WebExtension. Accessed: 21.10.2018.

[25] ——. (2018, September) Browser Extensions. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions. Accessed: 30.09.2018.

[26] ——. (2018, August) Content Security Policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP. Accessed: 02.01.2019.

[27] ——. (2018, September) Thunderbird/Add-ons Guide 63. https://wiki.mozilla.org/index.php?title=Thunderbird/Add-ons_Guide_63&oldid=1201285. Accessed: 21.09.2018.

[28] ——. (2019, January) WebExtensions. https://wiki.mozilla.org/index.php?title=WebExtensions&oldid=1200562. Accessed: 04.01.2019.

[29] NIST - National Institute of Standards and Technology, "Digital signature standard (dss)," Standard FIPS PUB 186-4, July 2013.

[30] B. Ramsdell and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification," RFC 5751 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–45, Jan. 2010. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5751.txt

[31] B. Ramsdell (Ed.), "S/MIME Version 3 Message Specification," RFC 2633 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–32, Jun. 1999, obsoleted by RFC 3851. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2633.txt

[32] ——, "," RFC 2898 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 18–19, Sep. 2000. [Online]. Available: https://tools.ietf.org/html/rfc2898#appendix-A.2

[33] ——, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specification," RFC 3851 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–36, Jul. 2004, obsoleted by RFC 5751. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3851.txt

[34] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979. [Online]. Available: http://doi.acm.org/10.1145/359168.359176

[35] Y. Tsiounis and M. Yung, "On the security of elgamal based encryption," in *Public Key Cryptography*, H. Imai and Y. Zheng, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 117–134.

[36] D. N. I. und Anwendungen (NIA), "Vertrauenswürdige systeme, die serversignaturen unterstützen - teil 2: Schutzprofil für qualifizierte signaturerstellungseinheiten zur serversignierung," DIN Deutsches Institut für Normung, Standard CEN/TS 419241-2, Juni 2017.

[37] ——, "Vertrauenswürdige systeme, die serversignaturen unterstützen - teil 1: Allgemeine systemsicherheitsanforderungen," DIN Deutsches Institut für Normung, Standard CEN/TS 419241-1:2018, September 2018.

[38] Wikipedia contributors, "Ciphertext indistinguishability — Wikipedia, the free encyclopedia," 2018, accessed: 01.01.2019. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Ciphertext_indistinguishability&oldid=875200011

[39] ——, "Elgamal signature scheme — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=ElGamal_signature_scheme&oldid=858566892, 2018, accessed: 28.10.2018.

List of Figures

2.1. High level overview of a TW4S (used free icon by IcoMoon.io - licensed as Creative Commons 3.0) 3
2.2. High level overview of a remote decryption service (used graphic by Marcio Duarte - licensed as Creative Commons 3.0) 4

3.1. System Architecture - Component diagram 9
3.2. File structure of addon 10
3.3. Communication flow between the add-on's components 11
3.4. ERD for TSP 12
3.5. ERD for PKS 13
3.6. User Registration Process 14
3.7. Key Generation Process for ElGamal 14
3.8. Key Generation Process for DSA 15
3.9. Decryption Process 16
3.10. Encryption Process 17
3.11. Multiple Common DSA Threshold Signing 18
3.12. Verification Process 19
3.13. Protocol flow of non-interactive ZKP 25
3.14. Example of attack against signing implementation with TSP 3 acting dishonest (used free icon by IcoMoon.io - licensed as Creative Commons 3.0) 26
3.15. Example of unsuccessful attack against decryption implementation with TSP 3 acting dishonest (used graphic by Marcio Duarte - licensed as Creative Commons 3.0) 27

A.1. Project schedule of conceptual, research and non-technical tasks 53
A.2. Project schedule of technical tasks 54
A.3. Milestones 55

C.1. Class Diagram Packets 68
C.2. Class Diagram Signature Subpackets 69
C.3. High level view on an OpenPGP public key 71
C.4. Explanation of field types used in following figures 71
C.5. Structure of an MPI in OpenPGP 71
C.6. OpenPGP packet header fields 72
C.7. OpenPGP packet tag in packet header on bit level 72
C.8. Structure of OpenPGP public key packet (tag 6) 72
C.9. Structure of OpenPGP user ID packet (tag 13) 73
C.10. Structure of OpenPGP signature packet (tag 2) 73
C.11. Bytes which are hashed for signature computation 74
C.12. High-level view on an encrypted OpenPGP message 75
C.13. Structure of OpenPGP public key encrypted session key packet (tag 1) 76
C.14. Structure of the decoded session key including the prefix 76
C.15. Structure of OpenPGP a symmetrically encrypted integrity protected data packet (tag 18) 76
C.16. Structure of OpenPGP a literal data packet (tag 11) 77

D.1. Add-on icon 87
D.2. Account 88
D.3. TSPs 88
D.4. PKS 88
D.5. Key Generation 89
D.6. My Keys 89
D.7. Public Key 90

D.8. Other's Keys 90
D.9. Context menu 91
D.10. Other's PGP Keys 91
D.11. Encrypted Text 92
D.12. Context menu 93
D.13. My Keys 93
D.14. Decrypted Message 93
D.15. My Keys 94
D.16. Decrypted Message 94
D.17. Verification Keys 95
D.18. Decrypted Message 95

List of Tables

A.1. Requirements specification 56
A.2. Requirements evaluation 60
A.3. Division of responsibilities 62

List of Listings

3.1. Multiple Common DSA Threshold Signature Schema 19

C.1. Trailer calculation in openpgp.js 75
C.2. Generating a safe prime using openssl 78
C.3. Ruby implementation details 79
C.4. Generating DSA params using Ruby 79
C.5. Initial post onto the mailing list 80
C.6. Confirmation by list moderator that question was posted 81
C.7. First reply subscriber of the mailinglist 81
C.8. Second reply by former CTO of PGP 82
C.9. Third reply by developer of Distributed Privacy Guard 82
C.10. Third reply by developer of Distributed Privacy Guard 83

APPENDICES

A. Project Management

This chapter briefly introduces our project management approach and gives an overview of the most important results.

A.1. Agile Method

We decided to use Scrum for the project management. Scrum is an agile project management framework that is convenient to deliver software. One of the most essential parts of Scrum are the sprints. In a sprint the team's focus is on certain tasks which are defined before the sprint. At the beginning of a sprint the team decides in a planning meeting, on which user-stories they want to work. Since we develop a proof of concept and in the beginning it was not completely clear what the end product will look like, this method enabled us to focus on the most significant deliverables from sprint to sprint.

Scrum can be adapted as it fits best for a certain project. We choose to work in sprints of 2 weeks. That matches best with the regular meetings with our tutors. We completely omitted the concept of daily standup meetings, since we synchronize on demand. Scrum uses the concept of user stories. User stories are the "work packages" we realize during a sprint. They consist of a short descriptor ("US-..."), a title ("I as a ") and optional description. In a user story we describe only very briefly what the task is all about, since we are only two people and work together very closely. We keep track of all our tasks using the project board provided by GitHub[1]. The progress of the project board is put to record in form of regular screen shots at the end of each sprint. The screen shots can be found in the digital content in appendix E.

A.2. Documentation

We try to keep the project management as slim and effective as possible. That is why we mostly use simple techniques to keep an overview and track the current status.

A.2.1. Meetings and Reviews

We meet with our tutors roughly every two weeks. Mostly on Friday, which is just before our sprint ends. The meeting is called "sprint review". We discuss open issues and show the progress we've achieved.

After each sprint we also reflect on the last two weeks: The progress we made, the up- and downsides and what we plan to do in the next sprint. This is the "sprint retrospective and sprint planning" meeting.

For each of these meetings we write a brief memo what we've discussed and decided. These memos are at the same time the documentation of decisions which are made during the project. However, early technical decisions are recoded in appendix B. There we explain why we use ElGamal or why we use Firefox as our client platform, just to name a few key decisions.

---

[1] `https://github.com/about` (accessed: 27.12.2018)

We do not add the meeting protocols here since we have around a thousand lines of meeting memos. That would just blow up this report. The memos can be found in the digital content in appendix E, though.

A.2.2. Deliverables

We did our best to show the output of the last sprint in the review meeting to our tutors. The deliverables overall differed a lot in their composition, some sprints were more related to conceptual and cryptography and others more focused on software and software architecture.

A.2.3. Requirements

We did a bit of a break with the agile approach with the requirements list we created at the beginning of the project. This list is used to fix the scope of the thesis with our tutors. The original list is an Excel file that can be found on the CD-ROM (see appendix E). In this document all these requirements can be found in section A.4 of this chapter.

A.2.4. Project schedule

As with the requirements we created a project schedule, which is not common in an agile approach. However, it was requested for this work and helped discussing the rough scope. In each sprint retrospective we updated the project schedule with the actual progress. We compare the estimated schedule with the actual progress in section A.3.

A.2.5. Source code

The source code we created in this project contains inline documentation. From the inline documentation we generated HTML files for better visualization of the code. We did not generate HTML documentation for the PKS component, because it is very simple code and HTML documentation would not provide any benefit. Since we used a Test-Driven-Development approach for the TSP and PKS components, we were able to measure our test coverage. We achieved 94.17% for the TSP and 98.94% test coverage for the PKS. Both the HTML documentation and the coverage reports are available on the CD-ROM in chapter E.

A.2.6. API

The API documentation was created using swagger[2]. It is available as HTML as digital content in chapter E.

A.2.7. Time sheet

We kept track of our work effort in a markdown file called "journal.md". It is included in the digital content in chapter E. The structure of the file is self-explanatory.

A.3. Course of the project

This section presents the course of the project based on the documentation we created during the project. We start with a review of the project schedule and look at the milestones.

---

[2] https://swagger.io/ (accessed: 08.01.18)

A.3.1. Project Schedule

At the beginning of the project we created a project schedule in an Excel file. After each sprint we tracked our actual progress. This file is not very nicely designed, so we created a prettily drawn project schedule using the same dates. The actual schedule is still available in digital form (see appendix chapter E). For better visualization we have split the schedule into two parts (figures A.1 and A.2). Mind that blue bars represent the planned tasks, gray bars optional planned tasks, yellow bars effective performance and red bars effective but unplanned performance.

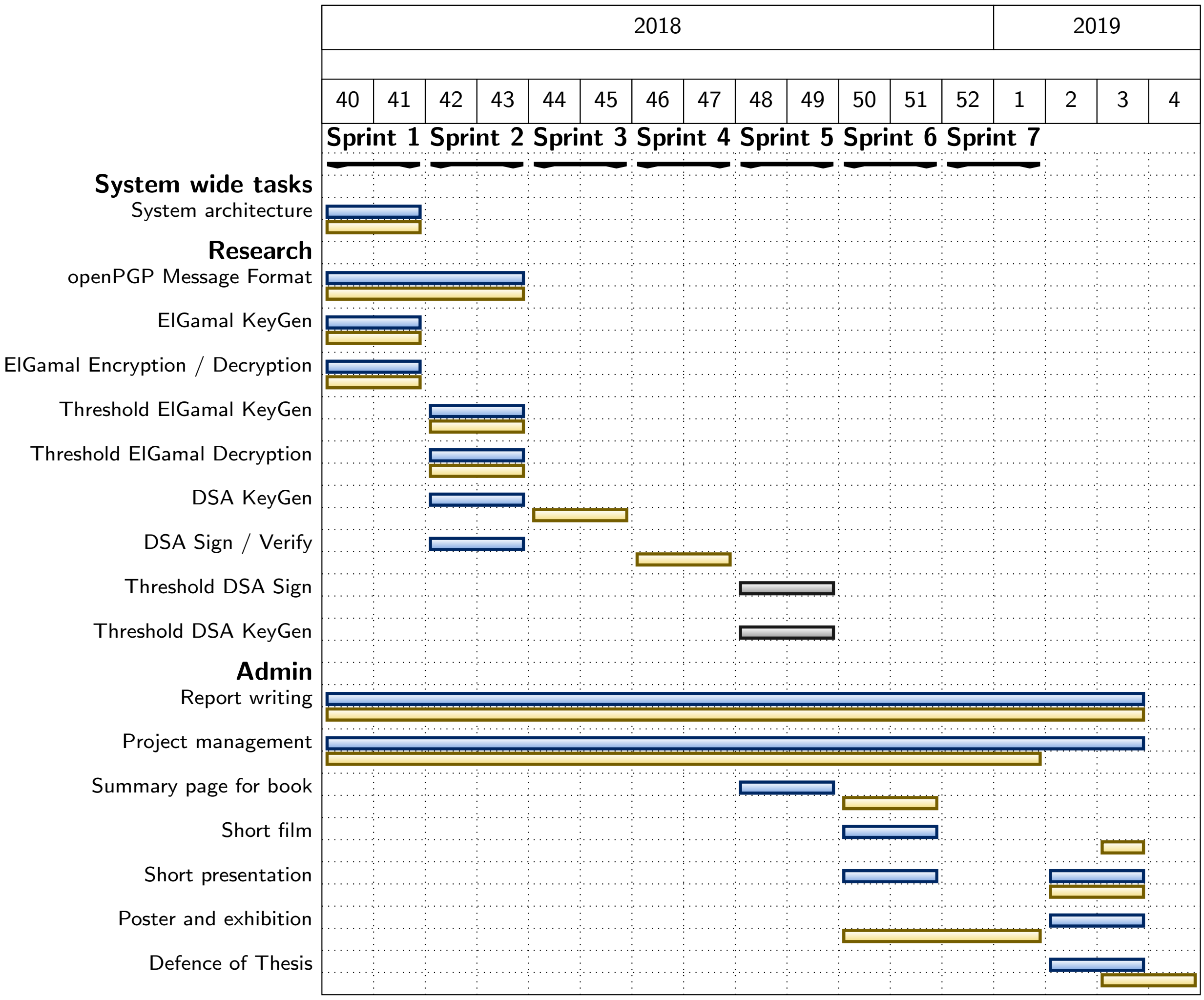


Figure A.1.: Project schedule of conceptual, research and non-technical tasks

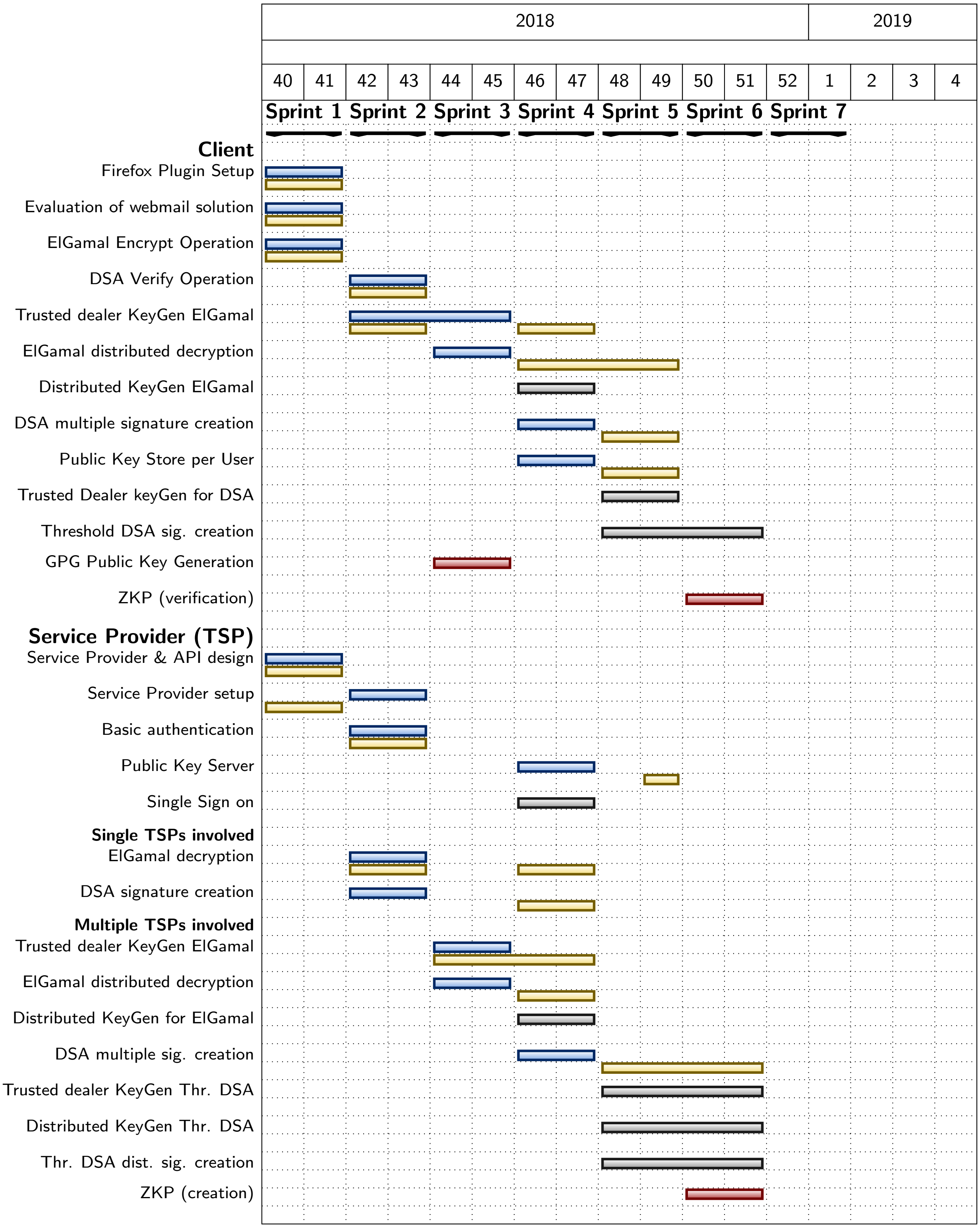


Figure A.2.: Project schedule of technical tasks

Looking at the project schedules shown in figures A.1 and A.2, it shows that the goal we initially aimed for has been very high. Subsequently we realized that dealing with OpenPGP used a lot more time than originally supposed. Unlike planned, we have implemented several parts of the OpenPGP RFC [9] instead of simply using a library for that. For this reason, during the last sprint we decided that we implement rather a cryptographic proof instead of dealing with threshold DSA. Threshold DSA could be a thesis of this scope for itself.

When it comes to administrative tasks, most tasks turned out to be a bit different. We continuously worked on the report and kept our project management up to date. However, we created the book entry later as planned and on the other hand we created the poster for the exhibition earlier than estimated. The short film is due on the 25th of January. For this reason we will create the film not before finishing this report. So we will create our presentation for the defense in the weeks 3 and 4.

A.3.2. Milestones

While we created the project schedule we have also defined a few milestones. Figure A.3 shows the estimated milestones in blue and the actual milestones in yellow. The shift between estimated and actual was caused by the unplanned task of implementing OpenPGP packets. Moreover, we did not strictly work on implementing the single TSP cases first. In the end, we managed to finish the implementation according to the planned deadline.

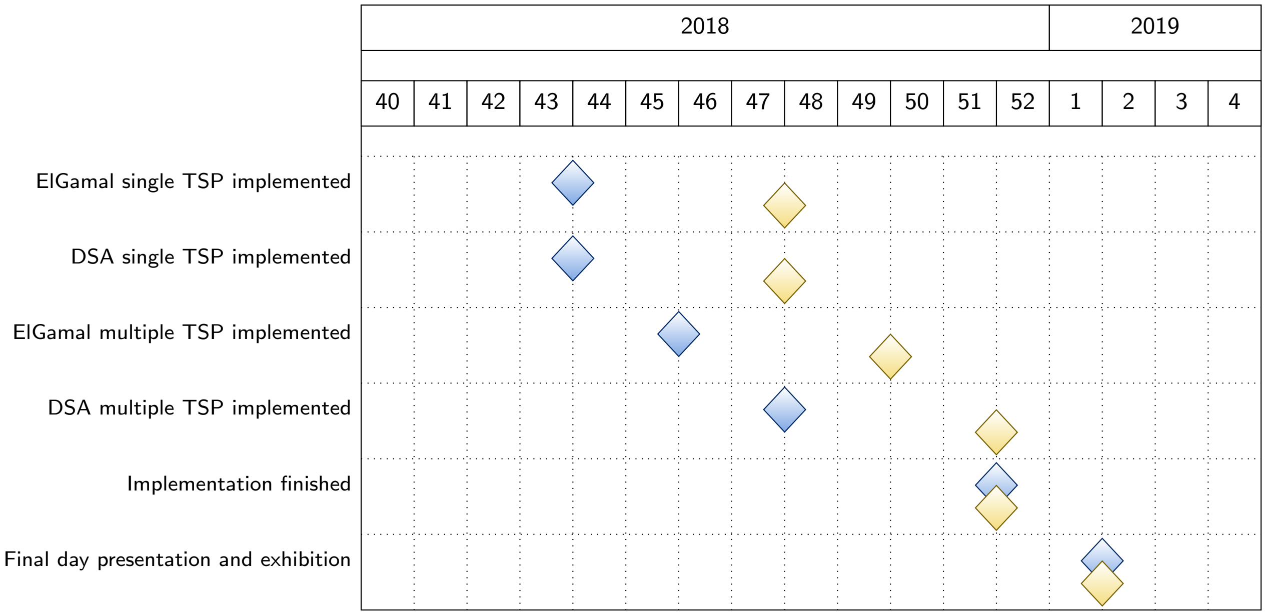


Figure A.3.: Milestones

A.4. Requirements

A.4.1. Requirements specification

The following table A.1 shows the requirements that have been defined at the beginning of the project.

Table A.1.: Requirements specification

ID	Title	Description	Priority	Acceptance creteria
FR	Functional Requirements			
FR.101	Encryption / Decryption			
FR.101.1	Common encryption / decryption operations			
FR.101.1.1	Message encryption	With our system it is possible to encrypt plaintext messages	REQUIRED	It is possbile to encrypt a plaintext in openpgp inline format (RFC4880) with our system
FR.101.1.2	Message decryption	With our system it is possible to decrypt plaintext messages	REQUIRED	It is possbile to decrypt a plaintext message that was encrypted in openpgp inline format (RFC4880) with our system
FR.101.2	Common encryption / decryption key generation			
FR.101.2.1	Key generation	With our System it is possible to create a valid key pair for encryption and decryption	REQUIRED	It is possible to create a valid 2048 bit private public key pair for encryption and decryption that can be used to cover requirement FR.101.1
FR.101.3	Threshold encryption / decryption operations			
FR.101.3.1	Threshold message decryption	With our system it is possible to decrypt plaintext messages in a threshold operation	REQUIRED	Threshold decryption of an inline encrypted plaintext works in a setup where the required number of TSPs deliver a sound response
FR.101.4	Threshold encryption / decryption key generation			
FR.101.4.1	Generate and distribute secret keys for decryption	With our system it is possible to generate a valid key pair that allows threshold decryption	REQUIRED	The user is able to create a new valid 2048bit key pair (using a trusted dealer model or a scheme that use distributed key generation among the TSPs), whereas the secret key is split into multiple shares for threshold decryption. The generated key can be used to cover requirement FR.101.3.
FR.101.4.2	Generate and distribute secret keys for decryption using distributed key generation	With our system it is possible to generate a valid key pair that allows threshold decryption, a distributed key generation scheme	OPTIONAL	The user is able to create a new valid 2048bit key pair (using a scheme that use distributed key generation among the TSPs), whereas the secret key is split into multiple shares for threshold decryption. The composed secret key for decryption never exists in the whole key generation process. The generated key can be used to cover requirement FR.101.3.
FR.101.5	Encryption / decryption third party compatibility			
FR.101.5.1	decryption compatibility with 3th party apps	Our client should be able to decrypt inline encrypted plaintext created by a third party application	OPTIONAL	It is possible to decrypt and display an inline encrypted plaintext email created with Enigmail (plantext - no markup, inline encrypted) for Thunderbird.

ID	Title	Description	Priority	Acceptance creteria
FR.101.5.2	encryption compatibility with 3th party apps	Our client should be able to inline encrypt plaintext of an email without attachments in a standardized format, compatible to a third party application	OPTIONAL	We can encrypt an email with our client (plaintext - no markup, inline encrypted) and it can be decrypted with Enigmail for Thunderbird
FR.102	Signing / Verification			
FR.102.1	Common signing / verification operations			
FR.102.1.1	Message signing	With our system it is possible to sign plaintext messages	REQUIRED	It is possbile to sign a plaintext in openpgp inline format (RFC4880) with our system
FR.102.1.2	Message signature verification	With our system it is possible to verify signatures of plaintext messages	REQUIRED	It is possbile to verify a plaintext message that was signed in openpgp inline format (RFC4880) with our system
FR.102.2	Common signing / verification operations			
FR.102.2.1	Message multi-signing	With our system it is possible to sign a plaintext message with multiple signatures	REQUIRED	It is possible to create multiple common signatures of a plaintext using the different keys at each TSP, which can be validated individually. The individual signatures may be grouped for collective verification. The format does not need to be compatible with any standard or third party application.
FR.102.2.2	Verify multple signatures	With our system it is possible to verifiy multiple signatures of a plaintext, created with different keys and decide if the result is accepted.	REQUIRED	It is possible to verify multiple signatures (created appriopriate to FR.102.2.1) of a plaintext, created using different secret keys and decide based on a threshold (how many singatures must be valid) if the result is accepted.
FR.102.3	Common signing / verification key generation			
FR.102.3.1	Key generation	With our System it is possible to create a valid key pair for creating and verifying signatures	REQUIRED	It is possible to create a 2048 bit private public key pair for creating and verifying signatures that can be used to cover requirement FR.102.1 and FR.102.2
FR.102.4	Threshold signing / verification operations			
FR.102.4.1	Threshold message signing	With our system it is possible to sign a plaintext message using a distributed cryptographic scheme	OPTIONAL	It is possible to creata a signature for a plaintext message, which is correctly signed using a threshold cryptographic scheme, whereas the shares of the private key (FR.102.5) are distributed among different TSPs
FR.102.4.2	Threshold signature validation	With our system it is possible to validate a signature which is created by a distributed cryptographic scheme	OPTIONAL	It is possbile to verify a sound signature of a plaintext message that was created using a threshold cryptographic scheme.
FR.102.5	Threshold signing / verification key generation			

ID	Title	Description	Priority	Acceptance creteria
FR.102.5.1	Generate and distribute secret keys for signing	With our system it is possible to generate a valid key pair that allows threshold signing	OPTIONAL	The user is able to create a new valid 2048bit key pair (using a trusted dealer model or a scheme that use distributed key generation among the TSPs), whereas the secret key is split into multiple shares for threshold signing.
FR.102.6	Signing / verification third party compatibility			
FR.102.6.1	verification compatibility with 3th party apps	Our client should be able to verify inline signatures of a plaintext created by a third party application	OPTIONAL	It is possible to verify an inline signed plaintext email created with Enigmail (plantext - no html, inline encrypted) for Thunderbird.
FR.102.6.2	signing compatibility with 3th party apps	Our client should be able to inline sign a plaintext of an email without attachments in a standardized format, compatible to a third party application	OPTIONAL	We can sign an email with our client using common signatures and its signature can be verified with Enigmail for Thunderbird
FR.102.6.3	signing compatibility with 3th party apps	Our client should be able to inline sign a plaintext of an email without attachments in a standardized format, compatible to a third party application	OPTIONAL	We can sign an email with our client using threshold signing and its signature can be verified with Enigmail for Thunderbird
FR.103	Local Environment			
FR.103.1	Configuration			
FR.103.1.1	The threshold can initially be defined by the user	The threshold for signing and decryption can initially be defined by the user at the time of key generation	RECOM-MENDED	If a new key pair for theshold signing or decryption is created, it is possible for the user to choose what threshold will be used for decryption or signing, respectively.
FR.103.1.2	The TSPs can be selected by the user	As a user it is possible to configure different TSPs at his own choice	REQUIRED	Before a new key pair is generated, the user can choose out of a given set, which TSPs he wants to use. There is a limit for the user of minimal and maximal number of TSPs that can be configured for a key pair.
FR.103.2	Architecture			
FR.103.2.1	client generic functionality	The use of the client is not restricted to specific website	OPTIONAL	It is possible for the client to interact with any visible HTML text area within the DOM of a website
FR.103.2.2	client portability	Major OS support	OPTIONAL	It is possible to use the client on Linux, Mac and Windows
FR.104	Remote Environment			

ID	Title	Description	Priority	Acceptance creteria
FR.104.1	Authentication			
FR.104.1.1	TSP Login	Every TSP needs a login mechanism	REQUIRED	It is possible for a user to authenticate against each TSP with at least a single factor.
FR.104.2	API			
FR.104.2.1	TSP cryptographic operations	The TSP must expose an API for cryptographic requirements FR.101 and FR.102	REQUIRED	The TSPs provide the cryptographic operations defined in (FR.101, FR.102) as a standardized API which triggers the following functions: 1. generate a key pair (FR.101.2, FR.101.4, FR.102.3, FR.102.5) 2. decrypt a message (FR.101.1, FR.101.3) 3. sign a message (FR.102.1, FR.102.2, FR.102.4) Whereas 2 and 3 may implement threshold cryptographic operations
FR.104.3	Architecture			
FR.104.3.1	E2E encrypted communication with TSP	The end users device and the service provider communication should be end to end encrypted.	OPTIONAL	The e2e communication is securely encrypted and authenticated between the client and the cryptographic component in addition to TLS
NFR	Non-functional requirements			
NFR.201	Non-functional requirements			
FR.201.1	Usability			
FR.201.1.1	Convenient Login Process	The login process for the user should be as convenient as possible.	RECOMMENDED	The user has the possibility to login once for each TSP and the the system should remember the user for a certain amount of time
FR.201.1.2	User documentation	The team must provide a small documentation for end users	RECOMMENDED	The installation of the plugin, key generation and at least one cryptographic operation from a user perspective are documented
FR.201.2	Documentation			
FR.201.2.1	System documentation	The key features are well documented	REQUIRED	At least the following documentation exists: • sequence diagrams • architecture • api design • system component diagram
FR.201.2.2	Database Documentation	The database structure is documented	REQUIRED	An entity-relationship diagram exists for the main database(s)
FR.201.2.3	Inline code documentation	The code is inline documented	RECOMMENDED	The code is inline documented according language best practices

A.4.2. Requirements evaluation

The requirements defined in the preceding table A.1 were initially fixed with the tutors. The following table shows the evaluation for each requirement. For the evaluation we used the terms “Yes”, “Yes*” and “No” to assess if the requirement was accomplished or not. The term ”Yes*“ means the requirement technically is fulfilled but that is at least one missing feature to present the functionality. Mind the correspondent comment for details.

Table A.2.: Requirements evaluation

ID	Achieved	Description
FR	Functional requirements	
FR.101	Encryption / Decryption	
FR.101.1	Common encryption / decryption operations	
FR.101.1.1	Yes	Encrypting plaintext using our add-on is compatible to OpenPGP achievable.
FR.101.1.2	Yes	Decrypting with our system can be achieved with a single TSP as well.
FR.101.2	Common encryption / decryption key generation	
FR.101.2.1	Yes	Generating a key pair using our add-on is possible. We use 2048 bit keys by default.
FR.101.3	Threshold encryption / decryption operations	
FR.101.3.1	Yes	Decrypting a plaintext using a threshold scheme with multiple TSPs is possible.
FR.101.4	Threshold encryption / decryption key generation	
FR.101.4.1	Yes	We implemented a general use case, which allows to select any number of participants and threshold in the scheme. 2048 bit key-sizes are hard-coded.
FR.101.4.2	No	Distributed key generation is not possible with our system.
FR.101.5	Encryption / decryption third party compatibility	
FR.101.5.1	Yes*	Decrypting a message encrypted by an OpenPGP compliant tool is possible, but not a message that was encrypted using Enigmail. Enigmail implicitly encrypts the message with the sender’s and the receiver’s public key. Our add-on is only able to handle one key. Using GNU Privacy Guard allows encrypting messages that our add-on is able to decrypt. Technically it is compatible since Enigmail uses the native GNU Privacy Guard binary installed on the client.
FR.101.5.2	Yes*	Technically it would be feasible to encrypt a message with an externally generated PGP public key, but we have not implemented the import of PGP public keys. Therefore this requirement may be not achieved. For testing purposes, we encrypted texts using PGP public keys generated with GnuPG, hence we know it would be possible.
FR.102	Signing / Verification	
FR.102.1	Common signing / verification operations	
FR.102.1.1	Yes	It is possible to sign a text compliant to the OpenPGP format.
FR.102.1.2	Yes	It is possible to verify a signature compliant to the OpenPGP format.
FR.102.2	Multiple Common signing / verification opreations	
FR.102.2.1	Yes	With our system it is possible to sign a plaintext using multiple trust service providers.
FR.102.2.2	Yes	With our system it is possible to verify a plaintext signed by multiple trust service providers.
FR.102.3	Common signing / verification key generation	
FR.102.3.1	Yes	Our system generates multiple DSA-2048 key pairs to accomplish key generation process for the multiple signature scheme.
FR.102.4	Threshold signing / verification operations	
FR.102.4.1	No	No, we have not implemented a scheme using secret sharing for signing.
FR.102.4.2	Yes*	Cannot be tested, because FR.102.4.1 was not accomplished. But a signature using a threshold scheme and secret sharing would generate a common signature, therefore we could verify the signature.
FR.102.5	Threshold signing / verification key generation	
FR.102.5.1	No	No, generating a key pair for signing whereas the private key is split into multiple pieces is not implemented.

ID	Achived	Description
FR.102.6	Signing / verification third party compatibility	
FR.102.6.1	Yes*	Technically it would be possible to verify an externally signed OpenPGP compliant message. Since we have not implemented the import of PGP public keys, this requirement is not fulfilled. For testing purposes we used externally generated PGP public keys to verify signatures.
FR.102.6.2	Yes	Signed messages by our add-on can be verified by external OpenPGP compliant tools like Enigmail or GnuPG.
FR.102.6.3	No	We have not implemented threshold signing using secret sharing, therefore this requirement is not accomplished.
FR.103	Local Environment	
FR.103.1	Configuration	
FR.103.1.1	Yes	Generating a threshold key pair for decryption and signing is possible with our add-on. The threshold can be defined by the user during the key generation process.
FR.103.1.2	Yes	The user can select the TSPs used for a certain key pair.
FR.103.2	Architecture	
FR.103.2.1	Yes	Our add-on can select text on a website or use an editable textarea element for cryptographic operations.
FR.103.2.2	Yes	The add-on is currently limited to Mozilla Firefox, but not limited to an operating system.
FR.104	Remote Environment	
FR.104.1	Authentication	
FR.104.1.1	Yes	The user authenticates itself on every TSP using a single factor (email, passphrase).
FR.104.2	API	
FR.104.2.1	Yes	The TSP exposes endpoints for generating a key pair (signing), partially decrypting a message and signing a message.
FR.104.3	Architecture	
FR.104.3.1	No	The messages are not additionally e2e encrypted nor authenticated between the add-on and the TSP.
NFR	Non-functional requirements	
NFR.201	Non-functional requirements	
NFR.201.1	Usability	
NFR.201.1.1	Yes	The user logs in once, and the add-on remembers the password. The credentials for the TSPs are derived, therefore they have the same lifetime.
NFR.201.1.2	Yes	A documentation how to install the plugin and how to execute cryptographic operations exists in chapter D
NFR.201.2	Documentation	
NFR.201.2.1	Yes	The documentation for the program flow, the architecture, the api design and the system components exist.
NFR.201.2.2	Yes	The relational databases are documented using a an entity-relationship diagram.
NFR.201.2.3	Yes	The code of the add-on, the TSP and PKS are inline documented.

A.5. Responsibilities

Table A.3 shows the division of responsibilities in this work. The one who is responsible for a certain chapter is not automatically the only author. Most of the chapters have been written by both students.

Table A.3.: Division of responsibilities

№	Chapter / Section	Responsibility
1	Introduction	Roger Ellenberger
2	Technical Background	Roger Ellenberger
3	Results	
3.1	System Architecture	Tobias Flühmann
3.2	Application flow	Tobias Flühmann
3.3	OpenPGP	Roger Ellenberger
3.4	Cryptography	Roger Ellenberger
3.5	Attack scenarios	Tobias Flühmann
4	Discussion	Tobias Flühmann
5	Conclusion	both
	Appendices	
A	Project Management	Roger Ellenberger
B	Evaulation	both
C	Implementation Details	
C.1	Add-on	Tobias Flühmann
C.2	Back-end	Tobias Flühmann
C.3	OpenPGP	Roger Ellenberger
C.4	Cryptography	both
D	User Documentation	Tobias Flühmann
E	Content of CD-ROM	Roger Ellenberger

Responsibilities of the created source code is documented in the inline comments. The tag "@author" describes who worked on the particular code. In JavaScript the tag is used per class whereas in Ruby it is used for each method by convention. The main author of a class or method comes first. Classes which have been developed entirely in pair programming and therefore both contributed equally, are tagged with "@author Created in pair programming".

B. Evaulation

B.1. Cryptography

B.1.1. Encryption schemes

The success of this project depends on the decision which cryptographic building blocks are used. In our preceding work [7] we looked at both Threshold RSA and Threshold ElGamal. We've seen that Threshold RSA is more complex than discrete logarithm based schemes. The main conclusion in [7] was, that we recommend using a DL based threshold scheme. We remain in the position we made in [7] and think it is easier to implement a robust system if the underlying scheme is simple.

B.1.2. Signatures schemes

Singing and encryption schemes in a system usually make use of the same mathematical problem. ElGamal is based on DL, so the signature scheme might also rely on DL. Moreover, Threshold RSA for signing suffers the same problems as for encryption. DSA is the obvious choice in this case. However, Threshold DSA is quite complex. But lately there has been progress in this field. Thanks to Bitcoin, which uses elliptic curve DSA, the topic got more research attention. In 2016 Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan proposed a threshold-optimal DSA/ECDSA scheme [10]. The scheme was also implemented, and a valid Bitcoin transaction has been signed using this scheme. For these reasons we have decided to use DSA as our main signature scheme.

B.2. email encryption and signature standards

There are three options in this project to use signing and encryption in email messages. Use the S/MIME standard, OpenPGP email signature and encryption, or create an own cryptographic format for signing and encryption. The main point about email standards is their cryptographic support which goes along with the cryptographic decisions in the previous section B.1.

S/MIME Common email clients such as Microsoft Outlook, Apple's standard email Client "Mail", or Mozilla Thunderbird natively support S/MIME. It requires a public key infrastructure (PKI) to validate the authenticity of the public key. This is an advantage in enterprise environments, where it is impractical to the end user to meet all recipients in person first and exchange the public keys or use a web of trust approach. In our case this advantage also hsa a downside. We would have been required to set up a PKI infrastructure for testing purpose. The other disadvantage of S/MIME is the lack of wide support of discrete logarithm based public key encryption schemes. It does support Diffie-Hellman (DH) to encrypt messages, but since version 3.0 of S/MIME [31] the support of it in the RFCs is falling. Other than in version 3.0, in the versions 3.1 [33] and 3.2 [30] DH is no more a must. S/MIME does not support ElGamal encryption at all.

OpenPGP The main advantage of OpenPGP [9] is the support for ElGamal encryption. Additionally, there are a lot of libraries and open source projects which already use OpenPGP. That is why the support of OpenPGP for common email clients is quite good. A lot of plug-ins such as "Enigmail" for Mozilla Thunderbird exist. OpenPGP supports RSA and DSA for signing messages.

In PGP applications the trust between parties is established by signing other people's keys. The idea behind this is called the web of trust. The web of trust is not a very practical idea. Thus, most people use direct trust and only use keys they verified directly by meeting someone in person or checking the fingerprint of a key they received over a second channel.

Decision Since OpenPGP natively supports the cryptographic schemes we plan to use, it suits us better for this project. Also, the trust model of PGP means less effort for implementing than the approach of S/MIME. We use OpenPGP for our proof of concept because it allows us to demonstrate the advantages behind the idea of this work more easily.

B.3. Client technology

One of the most crucial decisions in the project is how to deliver the product to the end users. There are several possibilities, such as a fat client application, a plug-in for a certain email application, a browser add-on or as an extension for a web-based email application. We compared each of the possibilities against their advantages and disadvantages in matters of dependencies, risk, existing knowledge of the platform and used programming language, operating system independence, and reachable end users. We only focused on the use-case of email decryption and signing in this section, since this was the scope of the project in the beginning. Later we have opened the scope to a generic use case and not only email security.

Fat client application The main problem with writing a fat client application is operating system dependencies. Either we would limit the final product to one operating-system, or we use a technology that allows us to write independent code. To develop an application that communicates with remote signing services is not an issue. The actual obstacle is the integration in the different platforms to promote a good user experience. We have only little experience with programming applications that have a lot of operating system specific dependencies.

Email application plug-in We could write a specific plug-in for a popular email application such as Mozilla Thunderbird[1] or Microsoft Office Outlook[2], since our use case is limited to emailing. Creating a specific plug-in allows deep integration in a client. On the other side building such a plug-in would restrict us to the application. For instance, if we would like to release it for both of the named email clients, we had to deal with two totally different programming languages.

We would like to support all of the major operating systems. Microsoft Office Outlook is not available for GNU/Linux. That's why we considered writing an extension for Mozilla Thunderbird as a possible option. However, Mozilla is currently in a transition of their plug-in architecture and developers of new extensions are in a undesired situation. The current version of Thunderbird at the point of the evaluation is 60.0. This version introduced changes for extensions. However, version 63.0 will again introduce new changes to the add-on architecture. Mozilla even addresses the current situation in their developer wiki.

> "The picture beyond Thunderbird version 60 is rather uncertain, given the continued high rate of change in Firefox platform code. We will work with add-on authors in the coming months to determine a viable post-60 add-on strategy." [27]

Browser add-on Another possibility would be to write a browser add-on that allows us to act as a client for email encryption and adds these functionalities to an existing web-mail client. A browser add-on allows to manipulate the document object model (DOM) of a website that is visited. Multiple browsers support extensions and even WebExtensions, but we focus on Mozilla Firefox[3] because it's the browser of our choice. Firefox uses the WebExtensions API for their add-ons [25]. With little effort such an extension could be ported to Mircosoft Edge[4]. A little more effort might be required to port such an add-on to Google Chrome[5][20].

---

[1] `https://www.thunderbird.net` (accessed: 29.09.2018)
[2] `https://products.office.com/en-us/outlook` (accessed: 29.09.2018)
[3] `https://www.mozilla.org/firefox/` (accessed: 29.09.2018)
[4] `https://www.microsoft.com/en-us/windows/microsoft-edge` (accessed: 29.09.2018)
[5] `https://www.google.com/chrome/` (accessed: 29.09.2018)

We already have experience with the technologies used to develop add-on's: "They are written using standard Web technologies - JavaScript , HTML, and CSS - plus some dedicated JavaScript APIs." [23].

It would be possible to introduce an architecture that allows other users to add support for other web mail clients, that we may not support from the beginning. Of course, this would allow us to easily add support for more web-mail clients if there is time at the end of our project.

Extension for web-based email applications A web-mail client can be hosted on an own server, for instance using one of the available open-source projects, such as Mailpile[6], Pixelated[7], or Roundcube[8]. Compared to the previous possibility of a browser add-on, this allows us to have full control over the client- and server-side code. The added client-side code could be integrated directly into the web interface without requiring installing an extension, what in certain circumstances can be an advantage. On the other hand, only few users maintain their own email server. Most users use a service operated by a provider where they have no control over the code on the server side. It might be possible that such an extension would be integrated by some providers, but the user has still not as much control of the code as by using an add-on.

Decision We consider developing a plug-in for Thunderbird as too risky for this project. Moreover, it would be a pity to develop something that will become legacy after just a few releases of the underlying application. A fat client or a plug-in for another email client like Microsoft Outlook requires technology we are not familiar with.

Both of us already have experience with web technologies. Thus, it makes sense either to create a browser add-on or an extension to a web-based email application. Developing an add-on for a web browser enables us to cover a wider range of use cases. Firefox is the browser of our choice, as already mentioned. So, we decided to create an add-on for Mozilla Firefox.

B.4. Back-end Technology

The decision about the back-end technology is less critical. There are not as many dependencies as on the client side. The back-end simply has to provide an API for the cryptographic operations. The state-of-the-art choice here is to use a RESTful API. Nearly any programming language that supports web services can be used. The more important part for us is the status quo of our knowledge of the very programming language and how good it supports our use case. The back-end mainly needs to support cryptographic operations, which means we are dealing with huge numbers. Especially in purely interpreted languages this should be implemented as efficiently as possible.

We considered using Python, Ruby, Java or JavaScript in the back-end. In the end the programming language that covers the mentioned points best is Ruby.

---

[6] `https://www.mailpile.is/` (accessed: 30.09.2018)

[7] `https://pixelated.github.io/` (accessed: 30.09.2018)

[8] `https://roundcube.net/` (accessed: 30.09.2018)

C. Implementation Details

C.1. Add-on

C.1.1. Communication between WebExtension components

Mozilla does not make a clear statement about communication between components of an add-on. They explain that there are two possibilities to exchange data but do not state in which scenarios to use which technology. Also, they do not provide a blueprint how the communication between the components should look like. The documentation for Google Chrome extensions is better in this case. Since Chrome extensions rely on the WebExtension standard as well the principle is adoptable. Based on [12, 11] we decided to use long lived connections which are called "ports" for most cases. In other cases, we use a technology that lets the developer use a background script in the front-end script context to execute a function located in the background scope. We are using this technique to interact from the pop-up and the options page with the public key server RESTful API to dynamically load the public keys [21]. Interacting with an API directly from the front-nd script does not make sense from a software architectures point of view and is prevented by default with the Content Security Possible (CSP) (can be bypassed).

C.1.2. UI framework

Bootstrap As a framework for the user interface we quite quickly came to the decision to use Bootstrap[1]. Bootstrap is one of the most popular front-end frameworks. We did not invest a lot of time in evaluating frameworks for front-end because we already knew Bootstrap and we quite like its default design.

C.1.3. Libraries

jQuery We mainly used plain JavaScript , but jQuery is used by Bootstrap. We used it in rare instances in code we wrote by ourselves.

openpgp.js OpenPGP.js[2] is the only well-maintained pure JavaScript library that we found by the time of starting this project. The library was completely audited two times by now. It supports ElGamal and DSA.

bn.js In cryptographic operations we work with huge integers (up to 2048 bits). We needed a library that enabled us to achieve this in JavaScript . OpenPGP.js makes use of bn.js[3] as well. We have chosen to use this library because it fulfilled our requirements as well. bn.js is able to perform computations in a reduced context in addition to simply represent big integers. This simplified calculations in certain groups a lot.

---

[1] `https://getbootstrap.com/` (accessed: 29.12.2018)
[2] `https://openpgpjs.org/` (accessed: 29.12.2018)
[3] `https://github.com/indutny/bn.js` (accessed: 29.12.2018)

C.1.4. OpenPGP Packets

In order to generate the packet structure described in section OpenPGP (C.3), we implemented each packet itself as the figure C.2 shows. Sometimes the implementation is really simple and is just setting a few values and return them into the super class.

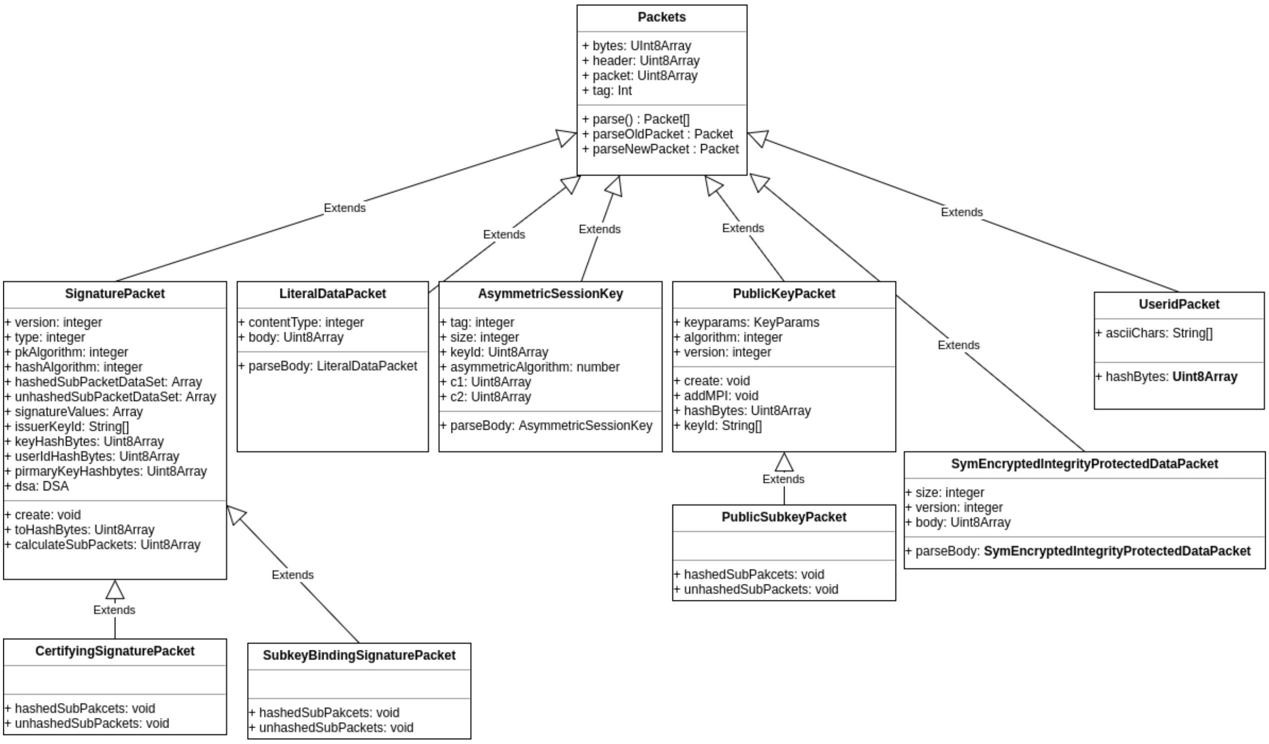

Packets
+ bytes: UInt8Array
+ header: Uint8Array
+ packet: Uint8Array
+ tag: Int
+ parse() : Packet[]
+ parseOldPacket : Packet
+ parseNewPacket : Packet
Extends
SignaturePacket
+ version: integer
+ type: integer
+ pkAlgorithm: integer
+ hashAlgorithm: integer
+ hashedSubPacketDataSet: Array
+ unhashedSubPacketDataSet: Array
+ signatureValues: Array
+ issuerKeyId: String[]
+ keyHashBytes: Uint8Array
+ userIdHashBytes: Uint8Array
+ pirmaryKeyHashbytes: Uint8Array
+ dsa: DSA
+ create: void
+ toHashBytes: Uint8Array
+ calculateSubPackets: Uint8Array
LiteralDataPacket
+ contentType: integer
+ body: Uint8Array
+ parseBody: LiteralDataPacket
AsymmetricSessionKey
+ tag: integer
+ size: integer
+ keyId: Uint8Array
+ asymmetricAlgorithm: number
+ c1: Uint8Array
+ c2: Uint8Array
+ parseBody: AsymmetricSessionKey
PublicKeyPacket
+ keyparams: KeyParams
+ algorithm: integer
+ version: integer
+ create: void
+ addMPI: void
+ hashBytes: Uint8Array
+ keyId: String[]
UseridPacket
+ asciiChars: String[]
+ hashBytes: Uint8Array
SymEncryptedIntegrityProtectedDataPacket
+ size: integer
+ version: integer
+ body: Uint8Array
+ parseBody: SymEncryptedIntegrityProtectedDataPacket
PublicSubkeyPacket
+ hashedSubPakcets: void
+ unhashedSubPackets: void
CertifyingSignaturePacket
+ hashedSubPakcets: void
+ unhashedSubPackets: void
SubkeyBindingSignaturePacket
+ hashedSubPakcets: void
+ unhashedSubPackets: void

Figure C.1.: Class Diagram Packets

On the other hand there are packets that are quite complex like e.g. the "SignaturePacket". The SignaturePacket has many fields, and it consists of multiple "SignatureSubPackets" as figure C.2 shows. The goal here was the same, to represent each packet separately and implement additional functionality if required. The "SignatureSubPackets" have a special property, they could be hashed or not. This property defines if the packet bytes are part of the hash that is signed in the end.

C.2. Back-end

For the back-end we used a development technique called Test-Driven-Development whenever possible. This technique allowed us to reduce implementation errors by first writing the test (using "rspec") and afterwards implementing the functionality fulfilling the test. In the back-end we primarily used dependencies already described in 3.1.2. There are a couple more dependencies we've used that mainly support the testing, documentation or the development experience described below. Since these dependencies include other dependencies themselves, a full list of dependencies can be found in the "Gemfile.lock" file for the TSP and PKS on the CD-ROM. The file is located in the root of the source code directory of each component.

- **grape**[4] Framework that simplifies creating APIs, providing a simple DSL.
- **grape_logging**[5] Define the logging style of the grape framework.
- **grape-swagger**[6] Simplify the usage of swagger together with the grape framework.

---

[4] `https://github.com/ruby-grape/grape` (accessed: 01.01.2019)
[5] `https://github.com/aserafin/grape_logging` (accessed: 01.01.2019)
[6] `https://github.com/ruby-grape/grape-swagger` (accessed: 01.01.2019)

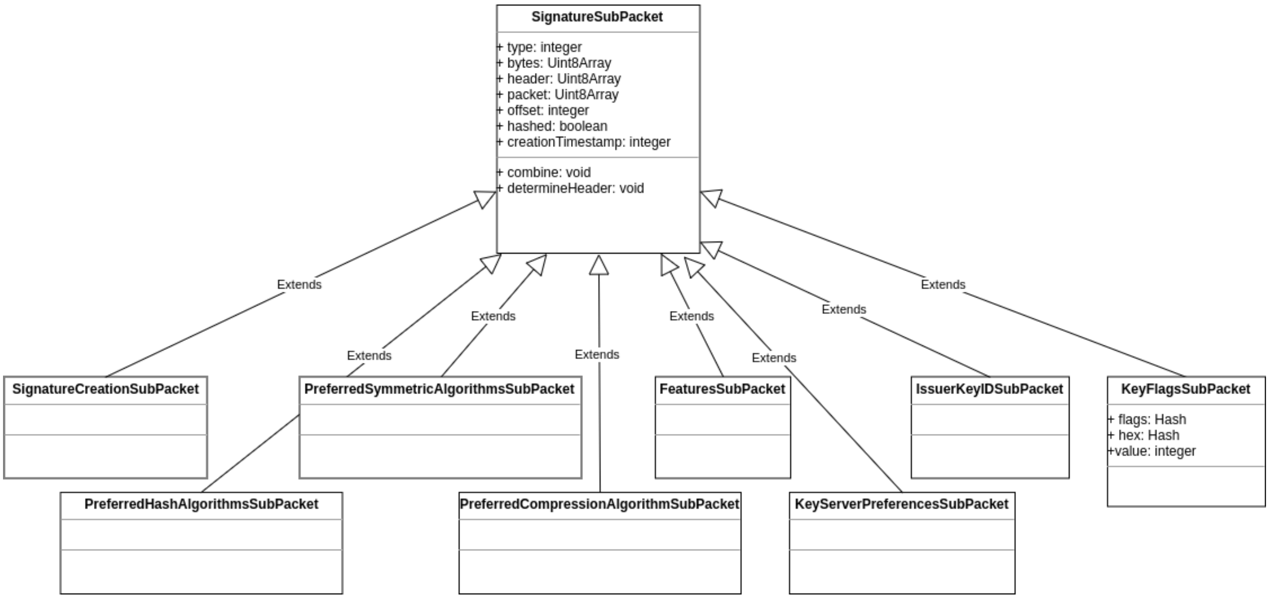

SignatureSubPacket
+ type: integer
+ bytes: Uint8Array
+ header: Uint8Array
+ packet: Uint8Array
+ offset: integer
+ hashed: boolean
+ creationTimestamp: integer
+ combine: void
+ determineHeader: void
Extends
SignatureCreationSubPacket
PreferredSymmetricAlgorithmsSubPacket
FeaturesSubPacket
IssuerKeyIDSubPacket
KeyFlagsSubPacket
+ flags: Hash
+ hex: Hash
+value: integer
PreferredHashAlgorithmsSubPacket
PreferredCompressionAlgorithmSubPacket
KeyServerPreferencesSubPacket

Figure C.2.: Class Diagram Signature Subpackets

- **activerecord**[7] ORM for Ruby (and Ruby on Rails).
- **activesupport**[8] Features for multibyte strings, I18n, time zones and testing.
- **pry**[9] Widely used ruby console, replacing irb.
- **rake**[10] CMake adaptation for running tasks written in Ruby.
- **pundit**[11] Simple and robust authorization with pure ruby classes.
- **mysql2**[12] Lightweight MySQL and MariaDB database adapter for Ruby.
- **sqlite3**[13] Small library that allows ruby programs to interact with sqlite3. Includes the sqlite3 database engine.
- **rspec**[14] Widely used behaviour driven development testing framework for ruby.
- **rspec-grape**[15] Extension of rspec to support grape framework testing.
- **database_cleaner**[16] Gem that simplifies testing with the database. It is possible to prepare, revert or truncate the database during test execution.
- **factory_bot**[17] Creating database objects using a factory. Multiple trait blocks are supported, additionally dynamic values, hashes and arrays can be generated.
- **faker**[18] Creating fake data, especially for testing purpose.
- **simplecov**[19] Analyze and display code coverage across multiple test-suites.
- **yard**[20] Code documentation generation tool.

---

[7] `http://rubyonrails.org/` (accessed: 01.01.2019)
[8] `http://rubyonrails.org/` (accessed: 01.01.2019)
[9] `http://pryrepl.org/` (accessed: 01.01.2019)
[10] `https://github.com/ruby/rake` (accessed: 01.01.2019)
[11] `https://github.com/varvet/pundit` (accessed: 08.01.2019)
[12] `https://github.com/brianmario/mysql2` (accessed: 11.01.2019)
[13] `https://github.com/sparklemotion/sqlite3-ruby` (accessed: 01.01.2019)
[14] `http://github.com/rspec` (accessed: 01.01.2019)
[15] `https://github.com/ktimothy/rspec-grape` (accessed: 01.01.2019)
[16] `http://github.com/DatabaseCleaner/database_cleaner` (accessed: 01.01.2019)
[17] `https://robots.thoughtbot.com/factory_bot` (accessed: 01.01.2019)
[18] `https://github.com/stympy/faker` (accessed: 01.01.2019)
[19] `https://github.com/colszowka/simplecov/` (accessed: 01.01.2019)
[20] `http://yardoc.org/` (accessed: 01.01.2019)

C.2.1. OpenPGP on the TSP

In the beginning of the project we found a Ruby library for handling the OpenPGP format. An initial idea was, to send received messages to each TSP, parse the packets, then partially decrypt the symmetric key and return it to the add-on. But with this setup, we would have sent too much information to the TSP and we decided only to send the very information needed to the TSP. Additionally, during our project we discovered that extracting messages and keys worked with this library, but some key functionality like extracting the symmetric key for ElGamal and key generation for ElGamal or DSA is missing. Therefore, we removed the library after finishing our own PGP implementation described in C.3. Some parts of our add-on implementation written in JavaScript are definitely inspired by the mentioned Ruby library[21].

C.3. OpenPGP

C.3.1. OpenPGP Keys

The add-on needs to be able to create PGP public keys on the client side, since we create private keys on the client, share it using a threshold scheme and inject it into the back-end via API call.

We have decided to use OpenPGP.js. This library provides ElGamal encrypt and decrypt operations and enables one to create and verify DSA signatures. However, it lacks the feature to create PGP keys for DSA and ElGamal. Openpgp.js internally uses JWA[22] to specify key parameters. JSON Web Algorithms (JWA), which do not provide support for neither ElGamal nor DSA. Because of the maintainer's decision to use JWA, we consider extending openpgp.js capabilities for our needs as a big effort. That is why we have decided to implement the assembly of a PGP public according RFC4880 [9] key from scratch.

We faced different challenges during the implementation. RFC4880 contains several quite complicated sections and lacks visualizations. Most of the packet format is described in prose only. In this section, we want to provide a deep dive explanation of the anatomy of a PGP public key. We start simple and get more and more into detail.

Packet types

The PGP public key we use is built using four different packet types.

- A public key packet
  The public key packet specifies mainly the used public key algorithm and the key parameters. This key is the main key in the PGP public key by convention and is used to certify other subkeys, which are included. Therefore, it must be a key which can be used for signing.
- A user ID packet
  This packet is rather simple. It is an UTF-8[23] string describing to whom the key belongs.
- Signature packets
  A signature packet is used to sign data. We create a self-signature of a key and a binding signature for each subkey. This packet is modular and makes use of subpackets. Typical information, which is included in a signature packet, is what the signed key may be used for, what symmetric encryption algorithms and hash algorithms a user wants to support, and most importantly the actual signature.
- Public subkey packets
  This packet has the exact same structure as a public key packet but uses another identifier to indicate that it is a subkey. A subkey may be a signing key or an encryption key.

A simplified visualization of a PGP public key may look like shown in image C.3. Mind that there are two signature packets. One self-signature over the public key packet and the user id packet and one binding signature over the public subkey packet.

All graphics that explain the packets on byte level, use the representation as explained in figure C.4.

---

[21] `https://github.com/dryruby/openpgp.rb`

[22] `https://tools.ietf.org/pdf/rfc7518` (accessed: 30.12.2018)

[23] `https://tools.ietf.org/pdf/rfc3629` (accessed: 14.11.2018)

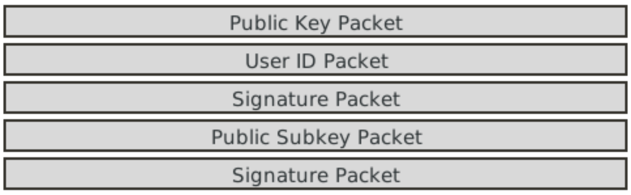

Public Key Packet
User ID Packet
Signature Packet
Public Subkey Packet
Signature Packet

Figure C.3.: High level view on an OpenPGP public key

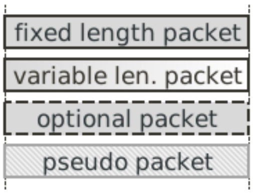

fixed length packet
variable len. packet
optional packet
pseudo packet

Figure C.4.: Explanation of field types used in following figures

Versions

Many of the packets, subpackets, headers and additional data structures make use of a version. Only version three and version four are supported for the components we use. An exception is the "symmetrically encrypted integrity protected data packet". The currently only version one defined for this packet is one. Version four is generally the recommended choice to generate new data if interoperability to old PGP versions is not an issue. Some implementations still use the old packet header format because of its simplicity.

Data element formats

Some fields in a PGP key have a defined format. We made use of the following:

- Scalar numbers
  A scalar number is either stored in one, two or four bytes in unsigned big-endian format.
- Multiprecision Integers (MPI)
  MPIs are used to store big numbers used in cryptographic algorithms. The first two bytes indicate the length of the MPI value in bits. The following bytes contain the value of the MPI. This field has a variable length. Figure C.5 visualizes the structure of an MPI.
- Time Fields
  This field is always four byte long and contains a UNIX-timestamp, which is the number of seconds that have passed since 1970-01-01 in UTC.

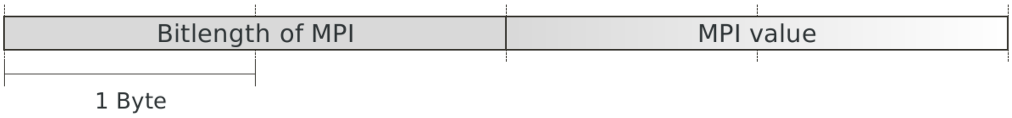

Bitlength of MPI
MPI value
1 Byte

Figure C.5.: Structure of an MPI in OpenPGP

Packet header

OpenPGP differs between two header formats; a version three and version four header. We decided to use a version three header because it is easier to implement and GnuPG[24] still uses this header type for newly generated keys. The old version three format starts with a one-byte tag and a variable length field.

---

[24] `https://gnupg.org/`

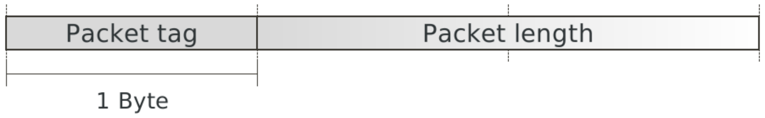

Packet tag
Packet length
1 Byte

Figure C.6.: OpenPGP packet header fields

The packet tag needs interpretation on the bit level. While the first bit by definition is always set to one, the second bit is in our case always set to zero, since we use a version three header. The packet type is a four-bit number. The length type (bits one and zero) define how big the following packet length field is:

0 The length field is one byte long. The packet is up to 255 bytes long.

1 The length field is two byte long. The packet is up to 65'535 bytes long.

2 The length field is four byte long. The packet is up to 4'294'967'295 bytes long.

3 There is no length field. The packet has an indeterminate length. We did not implement this case.

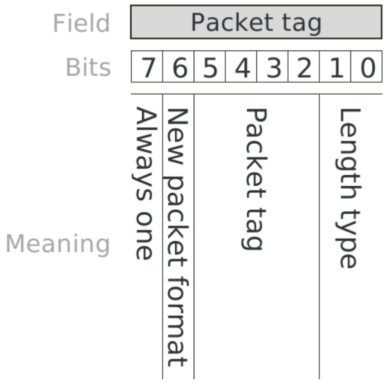

Field
Packet tag
Bits
7 6 5 4 3 2 1 0
Meaning
Always one
New packet format
Packet tag
Length type

Figure C.7.: OpenPGP packet tag in packet header on bit level

Detailed view on a PGP key

As shown in figure C.3 a PGP public key consists of several packets. Generally, each packet always has a header and a body. For simplicity, we only show the body of each packet.

A public key packet, which is the first packet in a PGP public key, is quite straightforward. After a version field, a four-byte UNIX timestamp of the creation time and the public key algorithm identifier come the public key parameters as MPIs. Figure C.8 shows a DSA key, which has four parameters. These are the parameters described in subsection C.4.1.

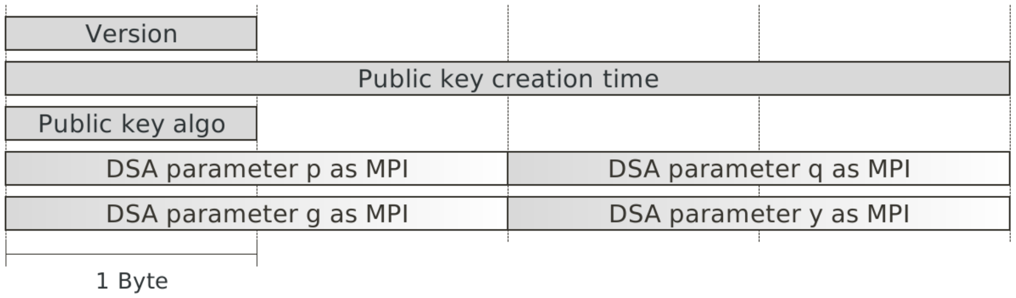

Version
Public key creation time
Public key algo
DSA parameter p as MPI
DSA parameter q as MPI
DSA parameter g as MPI
DSA parameter y as MPI
1 Byte

Figure C.8.: Structure of OpenPGP public key packet (tag 6)

The user ID packet is very basic as illustrated in figure C.9. It has only one field with variable length containing an ASCII string. As mentioned before, mind that this packet has a header as well that defines its length.

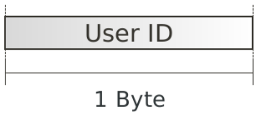

User ID
1 Byte

Figure C.9.: Structure of OpenPGP user ID packet (tag 13)

This signature packet is more complex in contrast to the user ID packet. The first field contains the version (we use version four). After that comes a one-octet signature type. In case of a signature over a public key, its value is either `0x10`, `0x11`, `0x12` or `0x13`. Few differences exist between the types. We have chosen `0x13` for our keys. The next two fields define the algorithm used for signing and hashing. After that comes a series of sub packets. Two types of sub packets exist: Hashed and unhashed ones. Hashed sub packets are included in the input for the signature creation. At the beginning of the hashed sub packets comes a two octet count specifying the sum of bytes of all hashed sub packets. Each sub packet is preceded with a length field, which has a variable length.

The first sub packet is the signature creation time, which is a four-byte UNIX timestamp. After that come a series of key properties each in an own subpacket. Key flags are used to specify what a key may be used for, symmetric algorithms and preferred hash algorithms specify used cryptographic primitives used for negotiating with the sender, and features describe additional capabilities of a key. We have chosen to support AES128, AES192 and AES256 as symmetric cipher and SHA256, SHA386 and SHA512 as preferred hash algorithms. All of them are state of the art at the time of writing. The last hashed sub packet describe key server preferences. In particular, it is one flag which specifies that the key may only be modified or updated by the user or the key server administrator.

After the hashed sub packets comes a series of unhashed subpackets, again starting with a two octet count with the number of bytes of all unhashed subpackets. Technically it is a series but in our case it is only one subpacket: The issuer's key ID. It is based on a SHA1 hash over a one octet value `0x99`, a two octet length of the public key packet and the entire body of the public key packet. The lower-order 8 bytes of that hash value represent the key ID.

After the key ID come the two left bytes of the key's checksum. How the checksum is formed is discussed in the next subsection, Signature creation. After these "hash bytes" the actual signature is appended as MPIs.

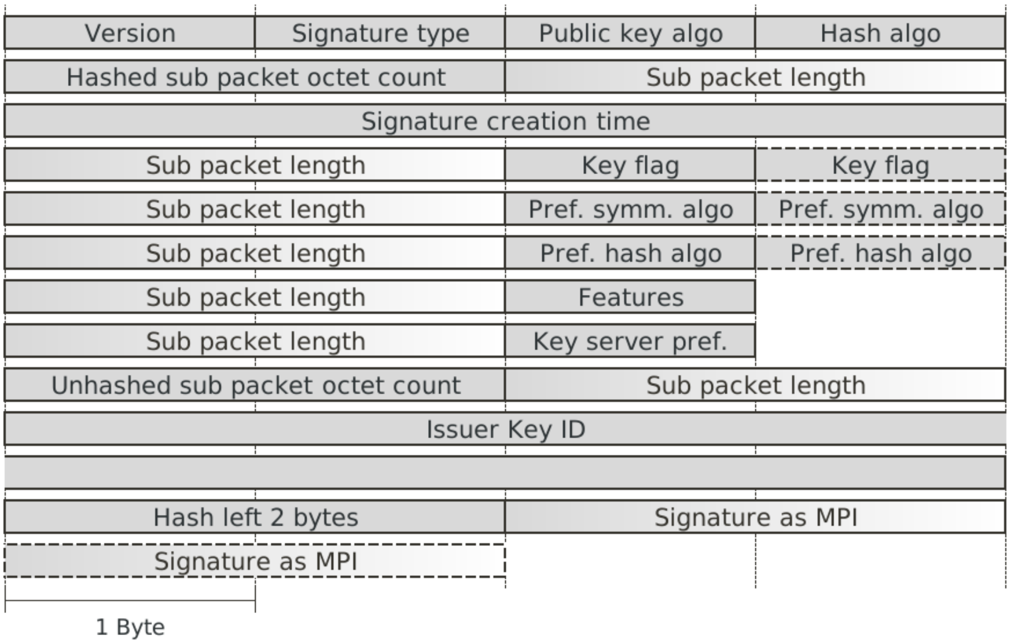

Version
Signature type
Public key algo
Hash algo
Hashed sub packet octet count
Sub packet length
Signature creation time
Sub packet length
Key flag
Key flag
Sub packet length
Pref. symm. algo
Pref. symm. algo
Sub packet length
Pref. hash algo
Pref. hash algo
Sub packet length
Features
Sub packet length
Key server pref.
Unhashed sub packet octet count
Sub packet length
Issuer Key ID
Hash left 2 bytes
Signature as MPI
Signature as MPI
1 Byte

Figure C.10.: Structure of OpenPGP signature packet (tag 2)

In case a key has additional public subkey packets, such a packet has the same structure as the public key packet shown in figure C.8. Only the header tag is different. Instead of tag `6`, the subkey packet has tag `14`. After the subkey packet comes another signature packet. The basic structure and it differs only a little. First the signature type is `0x18` ("subkey binding signature") instead of `0x13`. A subkey has less hashed subpackets. Namely, only the signature creation time and the key flags are included. Preferences on cryptographic primitives as symmetric ciphers and hash algorithms only appear in the signature over the public key packet. The key ID appears in the subkey as well, but it does not differ, since it is calculated over the "main" public key, not the public subkey. The remaining fields are similar to a public key packet.

Signature creation

Signature creation has several pitfalls. The bytes passed as input of the hash function are compound of data from the packets and "invisible fields" such as pseudo headers and a trailer. If there is an error in such a field, the resulting hash will be incorrect, without any hint where the mistake was made. In our case the implementation was quite time consuming, because the RFC provides no implementation examples. We found an example of the input data in an old thread of the OpenPGP mailing list [14]. With that byte-level values, we were able to find an implementation mistake in the trailer generation.

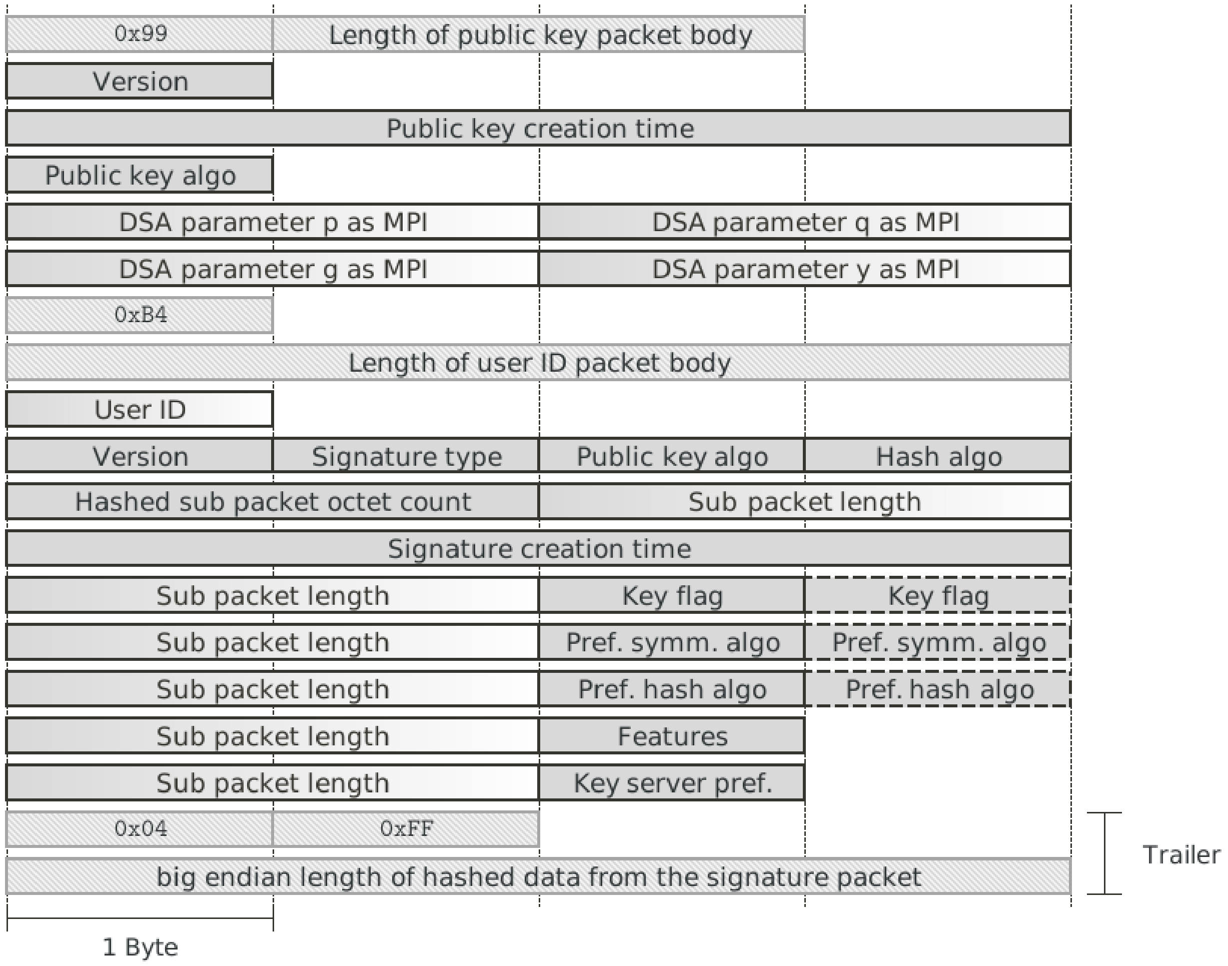

0x99
Length of public key packet body
Version
Public key creation time
Public key algo
DSA parameter p as MPI
DSA parameter q as MPI
DSA parameter g as MPI
DSA parameter y as MPI
0xB4
Length of user ID packet body
User ID
Version
Signature type
Public key algo
Hash algo
Hashed sub packet octet count
Sub packet length
Signature creation time
Sub packet length
Key flag
Key flag
Sub packet length
Pref. symm. algo
Pref. symm. algo
Sub packet length
Pref. hash algo
Pref. hash algo
Sub packet length
Features
Sub packet length
Key server pref.
0x04
0xFF
big endian length of hashed data from the signature packet
Trailer
1 Byte

Figure C.11.: Bytes which are hashed for signature computation

Figure C.11 shows the fields which are hashed for the signature computation and do not exist persistently. The shaded fields are generated only at time of signature computation. These are the pseudo headers for the public key packet and the user ID packet. They start with `0x99` and `0xB4`, respectively. Both have a fixed length field and the full packet body is hashed. Of the signature packets "the packet body starting from its first field, the version number, through the end of the hashed subpacket data" [9] is hashed. After that, a trailer is appended. It consists of a version number `0x04`, a fixed value `0xFF` and "big-endian number that is the length of the hashed data from the Signature packet" [9]. The trailer itself is not included in the length.

The value `0xFF` is some kind of magic value without any explanation. Looking at the implementation of OpenPGP.js we noticed that we are not the only ones who did not understand what that value stands for. On line six of listing C.1 the developers describe that very value with “?”.

```
1 Signature.prototype.calculateTrailer = function () {
2   let length = 0;
3   return stream.transform(stream.clone(this.signatureData), value => {
4     length += value.length;
5   }, () => {
6     const first = new Uint8Array([4, 0xFF]); //Version, ?
7     return util.concat([first, util.writeNumber(length, 4)]);
8   });
9 };
```

Listing C.1: Trailer calculation in openpgp.js

The length calculation of the trailer is where we made a mistake in our implementation. We only calculated the byte length of the hashed sub packets instead of the byte length of the body *until* the end of the hashed sub packets. We omitted to count six octets which resulted in a wrong hash.

C.3.2. OpenPGP messages

In addition to the creation of OpenPGP public keys we need to parse encrypted messages, since decryption works very differently in our case compared to OpenPGP.js. An encrypted message consists of several other packets in addition to the ones used for keys. While starting from scratch when assembling PGP public keys, implement parsing and decryption of messages was a lot simpler because we have already been familiar with RFC4880 at the time we implemented it.

Decryption of a PGP message

An encrypted message uses the following packet types:

- Public key encrypted session key packet
  This packet contains the symmetric key, which is asymmetrically encrypted using the recipient’s public key.
- A symmetrically encrypted integrity protected data packet
  The data in this packet is symmetrically encrypted with the key specified in the “public key encrypted session key packet”. It contains another packet. Either a compressed data packet or a literal data packet. We do not support compression.
- A literal data packet
  The literal data packet is rather simple. It contains a field describing how the data in the payload is formatted, a file name, a time stamp and the actual data.

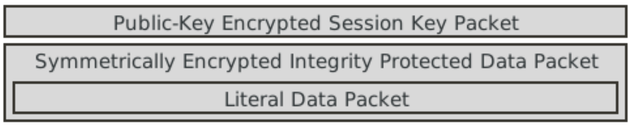

Public-Key Encrypted Session Key Packet
Symmetrically Encrypted Integrity Protected Data Packet
Literal Data Packet

Figure C.12.: High-level view on an encrypted OpenPGP message

Detailed view on a PGP encrypted message

PGP messages are put together from packets, which use the same Versions, Data element formats and C.3.1. We do not explain these again here. For details, refer to the corresponding subsection above.

Similar to the explanation in section Detailed view on a PGP key, we only describe the body of the individual packets. Mind that each packet has a header as explained in subsection C.3.1.

The first packet is a "public key encrypted session key packet". It starts with a version field, the used key ID of the recipient and an identifier of the used public key encryption algorithm. After that comes the actual encrypted session key. These are two MPIs in the case of ElGamal.

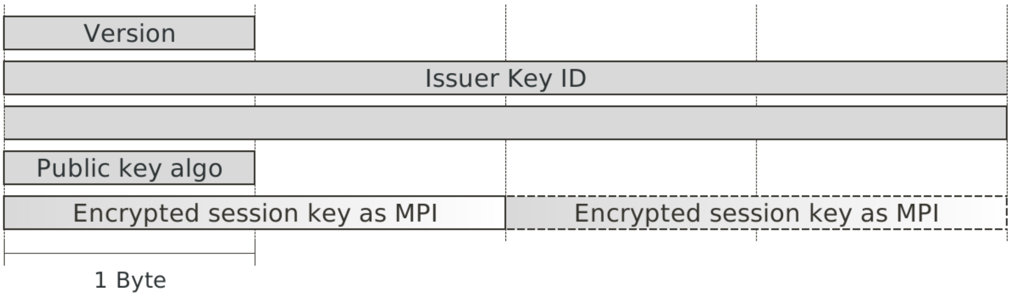


Figure C.13.: Structure of OpenPGP public key encrypted session key packet (tag 1)

The session key is EME encoded according PKCS #1 version 1.5 [16]. In decoded form, the key is composed of an identifier for the symmetric encryption algorithm, a checksum which is formed over the session key and the session key itself. The checksum allows the recipient to check easily if the decryption was successful. Figure C.14 illustrates the structure of the decoded session key with the additional prefix.

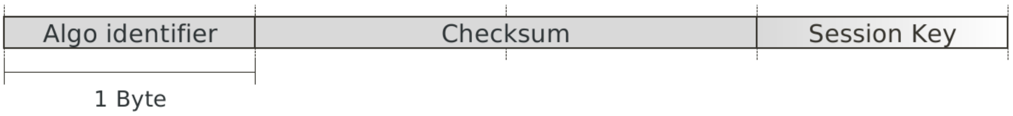


Figure C.14.: Structure of the decoded session key including the prefix

All the encrypted content of the message is stored in a "symmetrically encrypted integrity protected data packet" (tag 18). This is the new packet type, which included a modification detection code. Older implementation still use "Symmetrically Encrypted Data Packet" (tag 9), but we decided not to support that legacy type of packet. The new packet starts with a version field that is always set to one. The second field has a variable length and contains the encrypted data. Encryption is done using a special form of cipher feedback mode that uses resynchronization after the first block. We have not implemented that on our own, but used the implementation OpenPGP.js has provided. "The Modification Detection Code packet is appended to the plaintext and encrypted along with the plaintext using the same CFB context." [9].

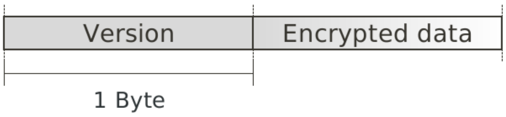


Figure C.15.: Structure of OpenPGP a symmetrically encrypted integrity protected data packet (tag 18)

Inside the symmetrically encrypted integrity protected data packet is a literal data packet (tag 11). As one of the only packets this has no version. It starts with a one-octet data format, where `0x62` stands for binary data, `0x74` for text and `u` for text formatted in UTF-8. The next field is the length indicator of its successor the file name field. After that is a four-octet UNIX timestamp which is associated with the literal data. "The remainder of the packet is literal data." [9] and obviously a variable length field.

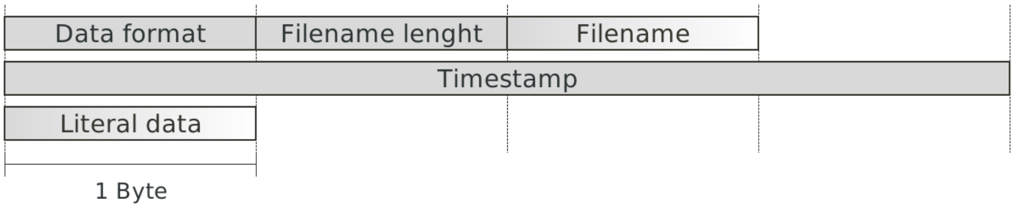


Figure C.16.: Structure of OpenPGP a literal data packet (tag 11)

C.4. Cryptography

C.4.1. Fixed groups

ElGamal

The client and the back-end use a safe prime as value p for all ElGamal operations. The value q is derived from p as a matter of course. Equation C.1 shows the calculation of q.

$$\begin{aligned}
p =& 25569964541394065091617251006653929414561869448354234099305430504898896633\\
& 70187978287315073194950317077167876330492809823811229754818645361493286663\\
& 73487825549377412390080136554659187049794230243714598392366947862029231201\\
& 05563765580858520886154961008124880595490852398995789578128745387315863\\
& 74728614140504107484186098174840541359693278008311652686781823884929886829\\
& 60852239143582790355319063245775546368486523936045598256506962529294502512\\
& 93378576864427714316753295737703268046010199327401297112543026122265386\\
& 35333879392741632940265925880036114122214400653311931174758482483991221521\\
& 29809946211607524014834192223763\\
q =& \frac{p-1}{2} = \\
& 12784982270697032545808625503326964707280934724177117049652715252449448168\\
& 50939891436575365974751585385839381652464049119056148774093226807466433318\\
& 67439127746887061950400682773295935248971151218572991961834739310146156\\
& 00527818827904292604430774805040624402977454261994978947890643726936579318\\
& 73643070702520537420930490874202706798466390041558263433909119424649434148\\
& 04261195717913951776595316228877731842432619680227991282534812646472512\\
& 56466892884322138571583766478688516340230050996637006485562715130611326931\\
& 76669396963708164701329629400180570611072003266559655873792412419956107606\\
& 4904973105803762007417096111881
\end{aligned} \tag{C.1}$$

ElGamal uses two groups: $\mathbb{Z}_p^*$ and the $\mathbb{G}_q$ which is a subgroup of the first one. We need a generator g for generating all the elements in $\mathbb{G}_q$. We find this generator by performing the calculations shown in equation C.2. The smallest generator of $\mathbb{G}_q$ is $g = 3$.

$$\begin{aligned}
g^q &\mod p \overset{?}{=} 1\\
2^q &\mod p \neq 1\\
3^q &\mod p = 1
\end{aligned} \tag{C.2}$$

We have computed the safe prime p using openssl[25].

Listing C.2: Generating a safe prime using openssl

```
$ openssl prime -generate -bits 2048 -safe -hex
CA8D9A507290306C39C9E34C0B469D4198E9C8F4EB8C105F089505EA308C7CC94B6CBC2AA8
944D886B3CDD21853651C742C4C3B016608E9E5AFA18D0BA654F886E462CFB46CE470986A7
C8982C9DFFC4CD1AA4026D47F3FF28898AE75C1539B828E13C840FF58D6A744E472E564EA8
FD14800564F35B824556155F8F9634D9EF7777927486816 1E942FD1D3639C3510ACEA33021
CF7471BD0B77EA7FE9EBFBCB5C051D65513C61FEDBBB32003E51C56B97A2F021947B70685C
9BA0F51716FF52ACD94DF96C4DF84C539AD86F1F40AD7C5CFEC8E98DAA3C34E7591DDB5013
DE2075FD4DE69956D5EFA7AA37B69AB06591D48043FB4555005D23C71177DF0C0E13
$ openssl version
OpenSSL 1.0.2p  14 Aug 2018
```

DSA

We decided to use groups with a 2048 bit length in our project. According to the NIST's DSS standard (cpater 4.1)[29], the group parameters have to be chosen as follows:

L bit length of p (2048)

N bit length of q (256)

p a prime modulus, where $2^{L-1} < p < 2^L$

q a prime divisor of $(p-1)$, where $2^{N-1} < q < 2^N$

$\hat{\mathbf{g}}$ a generator of a subgroup of order q in the multiplicative group of GF(p), such that $1 < \hat{g} < p$.

Finally we have used the following group parameters, generated using the commands in listing C.4.

$$
\begin{aligned}
p =&17408767115362160767854512331871817547545372810375504040633625800365550 08\\
&3522678433763525783349713183615874249694923551090347939317822052933550308\\
&5306956810558201673960871073558956191677024986071729929651868173363822875\\
&1314605820702681766266061541209369651494901713202164718444341970256881018\\
&5082459800588792648681099184450980374948942460710590968933830882661008798\\
&7568446135114784082777487416041375868787492397685505879896159814589789083\\
&7538897671049458695246629705310605326191464229988200661376577190094109775\\
&6663190566715490973479536686538472761901188424006559599378089903944070981\\
&38877377104023971905873671467 6639\\
q =&7049748588409177309781326394427777758977730269814649788554177311104991775\\
&3153\\
g =&3787163221898410052636428052669975178747256829698497858735807824346712708\\
&7526449397546720012906964228545328312829641711302981602776742518191483030\\
&6838309657639510087403944404782585546808701260176159884240219337644728766\\
&3443437576456792489512299406456761711343619101000804928455167324693622832\\
&7909857400305300399651253151811118323347751825309039116646748584806065507\\
&4206888619778071453999189671162434100365293550747893744818919606940666816\\
&4129804843587578665292148402059208404883053134318811464724210781078347441\\
&2965325497548796586759797178798929366045993249091797349775958232063500035\\
&896907035024783210661254586357 38
\end{aligned}
\tag{C.3}
$$

---

[25] `https://www.openssl.org/` (accessed: 29.11.2018)

We have generated the DSA parameters using the openssl implementation of Ruby (see listings C.3 and C.4).

Listing C.3: Ruby implementation details

```
$ ruby -v
ruby 2.5.1p57 (2018-03-29 revision 63029) [x86_64-darwin16]
$ ruby -r openssl -e "puts OpenSSL::VERSION"
2.1.0
```

Listing C.4: Generating DSA params using Ruby

```
$ irb -r openssl
irb(main):001:0> openssl = OpenSSL::PKey::DSA.new(2048)
=> #<OpenSSL::PKey::DSA:0x00007fe9da8844a8>
irb(main):002:0> openssl.p.to_s
=> "89e76c7ef274d3a72aa04d872ba8345..."
irb(main):003:0> openssl.q.to_s
=> "9bdc29ba7e23780f64af83cb1e4cb77..."
irb(main):004:0> openssl.g.to_s
=> "1e00074e4e8b97d137effa911c6b3c7..."
irb(main):005:0> OpenSSL::VERSION
=> "2.1.0"
```

C.4.2. Use of ElGamal in OpenPGP

OpenPGP's ElGamal implementation does not map the plaintext message into the prime order subgroup $\mathbb{G}_q$ but in $\mathbb{Z}_p$. IND-CPA security for ElGamal holds only in $\mathbb{G}_q$. Instead, OpenPGP uses EME encoding of PKCS #1 version 1.5. Although EME encoding was defined for RSA.

Neither in the RFC nor in the mailing list's archive exist any reference why this was designed that way. To find out more, we wrote our own question on the OpenPGP mailing list (see listings C.5 through C.10).

Background about ElGamal in PGP The community very quickly gave us valuable background information back the days PGP was designed and an appropriate statement about the security of ElGamal as it is defined in OpenPGP. The most substantial response regarding the background information came from Jon Callas[26]. He is the former CTO of PGP and was quite closely involved back then when these design decisions have been made. Jon Callas called Derek Atkins in the discussion, who was originally involved in PGP's design considerations.

According to Derek Atkins, ElGamal was introduced mainly "[...] to work around the RSA patents." [5] Jon Callas wrote, "The RSA patent expired in the year 2000, but the discrete log patents expired in '97. Thus, there was a real reason for wanting a discrete log option. They picked Elgamal because you can use it more or less as if it were RSA." [5].

ElGamal CPA security "OpenPGP is pretty much the first substantial protocol to use ElGamal keys" [5]. At the time the research have not yet defined semantic security for ElGamal. In 1998 Yiannis Tsiounis and Moti Yung published their work about ElGamal's security [35]. Heiko Stamer wrote on the mailing list that he "stumble[d] upon the same problem [...] when creating DKGPG [...]" [5]. DKGPG[27] is an OpenPGP implementation for Linux that uses distributed key generation and threshold cryptography. In his work he dug a bit deeper concerning this issue and pointed out:

> "Computing $m \in \mathbb{Z}_p^*$ from given g, y, g^k, $g^k m \in \mathbb{Z}_p^*$ is hard, iff the Computational Diffie-Hellman (CDH) problem is hard ElGamal in $\mathbb{Z}_p^*$ is OW-CPA secure under CDH assumption" [15]
>
> "Distinguishing $m, \bar{m} \in G_q$ given g, y, g^k, $g^k m$, $g^{\bar{k}}$, $g^{\bar{k}} \bar{m} \in G_q$ is hard, iff the Decision Diffie-Hellman (DDH) problem is hard ElGamal in G_q is IND-CPA secure under DDH assumption" [15]

---

[26] `https://en.wikipedia.org/w/index.php?title=Jon_Callas&oldid=864387606` (accessed: 09.12.2018)

[27] `http://nongnu.org/dkgpg/` (accessed: 01.01.2019

Additionally he wrote that in his opinion “it cannot be solved without revising ElGamal in the RFC.” [5] To sum up, we can say that OpenPGP does not use a semantically secure implementation of ElGamal.

Listing C.5: Initial post onto the mailing list

```
From: rogerandrea.ellenberger@students.bfh.ch
To: openpgp@ietf.org
Cc: tobias.fluehmann@students.bfh.ch
Subject: IND-CPA security of OpenPGP’s ElGamal implementation
Date: Sat, 8 Dec 2018 16:16:59 +0100

Hi,

We are currently working on a Firefox extension which allows us to
encrypt and decrypt OpenPGP messages using partially-trusted remote
keystores. We perform the asymmetric decryption operations on these
remote keystores using a threshold ElGamal scheme. We develop this
extension as our thesis project at Bern University for Applied Sciences.

ElGamal is our preferred asymmetric algorithm since we make use of
threshold cryptography and the threshold scheme of ElGamal is much
simpler compared to threshold RSA.

Working on the decryption of OpenPGP messages we saw in the section 5.1
of RFC4880 that the symmetric key $s$ with algorithm identifier and
checksum is encoded and padded using PKCS#1 v1.5 EME encoding. The
result of the EME encoding $eme_encode(s) = m$ then is encrypted using
ElGamal.

A short recap of ElGamal: We use the Groups Z*_p and G_q for ElGamal,
where p is a safe prime, $q = (p-1)/2$ and $G_q$ is a subgroup of
$Z*_p$. Encryption is defined as follows:

$
enc(y, r, m):   G_q x Z_q x G_q -> G_q x G_q
             :=  (c_1, c_2) = (g**k mod p, m * y**k mod p)
$

According literature one need to map $m$ into $G_q$ to guarantee that
ElGamal ist IND-CPA secure. According the RFC this check is not
performed, but an encoding/padding is applied to the plaintext.

Long story short: We would like to know what the considerations have
been to use ElGamal combined with a PKCS-EME encoding, since without the
encoding/padding it actually lacks of CPA security. Unfortunately we did
not find any authoritative reference which give a statement about
ElGamal security when $m$ is not in G_q but padded.

Thank you very much in advance for your help.


Best regards,
Tobias and Roger
```

Listing C.6: Confirmation by list moderator that question was posted

```
From: stephen.farrell@cs.tcd.ie
To: rogerandrea.ellenberger@students.bfh.ch
Date: Sat, 8 Dec 2018 15:25:56 +0000
Subject: Re: [openpgp] IND-CPA security of OpenPGP's ElGamal implementation

offlist, since I don't have a substantive answer for you...

Hopefully you get good answers, but if not, you may get better
ones from cfrg@irtf.org. I'd say wait a few days to see and if
you don't get a good answer then try asking on the CFRG list.

Cheers,
S.
```

Listing C.7: First reply subscriber of the mailinglist

```
From: Watson Ladd <watsonbladd@gmail.com>
To: rogerandrea.ellenberger@students.bfh.ch
Cc: IETF OpenPGP <openpgp@ietf.org>, tobias.fluehmann@students.bfh.ch
Subject: Re: [openpgp] IND-CPA security of OpenPGP's ElGamal implementation
Date: Sat, 8 Dec 2018 10:42:56 -0800

On Sat, Dec 8, 2018 at 7:21 AM Roger Ellenberger
<rogerandrea.ellenberger@students.bfh.ch> wrote:
><snip>
> Long story short: We would like to know what the considerations have
> been to use ElGamal combined with a PKCS-EME encoding, since without the
> encoding/padding it actually lacks of CPA security. Unfortunately we did
> not find any authoritative reference which give a statement about
> ElGamal security when $m$ is not in G_q but padded.

I suspect the only leak is one bit, namely whether or not m is a
quadratic residue mod p or not. The impact on the security is minimal.
But why not use threshold IECS as https://tools.ietf.org/html/rfc6637
does and avoid this quirk?

>
> Thank you very much in advance for your help.
>
>
> Best regards,
> Tobias and Roger
>
> _______________________________________________
> openpgp mailing list
> openpgp@ietf.org
> https://www.ietf.org/mailman/listinfo/openpgp



--
"Man is born free, but everywhere he is in chains".
--Rousseau.
```

Listing C.8: Second reply by former CTO of PGP

```
From: Jon Callas <joncallas@icloud.com>
To: Roger Ellenberger <rogerandrea.ellenberger@students.bfh.ch>
Cc: Jon Callas <joncallas@icloud.com>, openpgp@ietf.org, =?utf-8?Q?Fl=C3=
    BChmann_Tobias?= <tobias.fluehmann@students.bfh.ch>
Subject: Re: [openpgp] IND-CPA security of OpenPGP’s ElGamal implementation
Date: Sat, 8 Dec 2018 16:44:25 -0800

> On Dec 8, 2018, at 7:16 AM, Roger Ellenberger <rogerandrea.ellenberger@st=
udents.bfh.ch> wrote:
>
> [...]
>
> Long story short: We would like to know what the considerations have
> been to use ElGamal combined with a PKCS-EME encoding, since without the
> encoding/padding it actually lacks of CPA security. Unfortunately we did
> not find any authoritative reference which give a statement about
> ElGamal security when $m$ is not in G_q but padded.
>
> Thank you very much in advance for your help.

What a wonderful idea. I really like it.

The short answer to your question is that there aren’t any references. I
concur with Watson that any issue is minimal, especially if you’re using
Elgamal keys that are 3K to 4K.

A longer answer is that OpenPGP is pretty much the first substantial protocol
to use Elgamal keys. Back in the late ’90s, when patents were an issue, the
then "PGP 3" system which became "PGP 5" and then that became standardized as
OpenPGP, wanted an alternative to RSA. The RSA patent expired in the year 2000,
but the discrete log patents expired in ’97. Thus, there was a real reason for
wanting a discrete  log option. They picked Elgamal because you can use it
more or less as if it were RSA. As time went on, Elgamal signatures fell by
the wayside over DSA, leaving Elgamal for encryption.

Derek Atkins might remember more, because a lot of those original decisions
were made by some combination of him, Colin Plumb, and Hal Finney.

        Jon
```

Listing C.9: Third reply by developer of Distributed Privacy Guard

```
From: Heiko Stamer <HeikoStamer@gmx.net>
To: openpgp@ietf.org
Subject: Re: [openpgp] IND-CPA security of OpenPGP’s ElGamal implementation
Date: Sun, 9 Dec 2018 10:02:19 +0100

At 08.12.18 / 16:16 Roger Ellenberger wrote:

> According literature one need to map $m$ into $G_q$ to guarantee that
> ElGamal ist IND-CPA secure. According the RFC this check is not
> performed, but an encoding/padding is applied to the plaintext.

I stumble upon the same problem [1] when creating DKGPG some months ago.
In my oppinion it cannot be solved without revising ElGamal in the RFC.
Maybe the work of Sakurai and Shizuya [2] can help to understand some
implications of the RFC authors choice at that time.
```

```
[1] slide #17 of https://www.nongnu.org/libtmcg/dg81_slides.pdf
[2] https://link.springer.com/content/pdf/10.1007/3-540-49264-X_28.pdf

Best regards,
Heiko.
```

Listing C.10: Third reply by developer of Distributed Privacy Guard

```
From: Derek Atkins <derek@ihtfp.com>
To: Jon Callas <joncallas=40icloud.com@dmarc.ietf.org>
Cc: Roger Ellenberger <rogerandrea.ellenberger@students.bfh.ch>, =?utf-8?Q?Fl=
   C3=BChmann?= Tobias <tobias.fluehmann@students.bfh.ch>, <openpgp@ietf.org>,
    Jon Callas <joncallas@icloud.com>
Subject: Re: [openpgp] IND-CPA security of OpenPGP's ElGamal implementation
Date: Mon, 10 Dec 2018 11:21:18 -0500

Jon Callas <joncallas=40icloud.com@dmarc.ietf.org>; writes:

> A longer answer is that OpenPGP is pretty much the first substantial
> protocol to use Elgamal keys. Back in the late '90s, when patents were
> an issue, the then "PGP 3" system which became "PGP 5" and then that
> became standardized as OpenPGP, wanted an alternative to RSA. The RSA
> patent expired in the year 2000, but the discrete log patents expired
> in '97. Thus, there was a real reason for wanting a discrete log
> option. They picked Elgamal because you can use it more or less as if
> it were RSA. As time went on, Elgamal signatures fell by the wayside
> over DSA, leaving Elgamal for encryption.
>
> Derek Atkins might remember more, because a lot of those original
> decisions were made by some combination of him, Colin Plumb, and Hal
> Finney.

As I recall, we added DSA and ElGamal to attempt to work around the RSA
patents.  For a while, IIRC, PGP3/5 did not even support RSA at all,
making it harder to interact with PGP2 -- at least for a short while.
This was all back in 1996-1997 when this work was done.

>       Jon

-derek
--
       Derek Atkins                 617-623-3745
       derek@ihtfp.com             www.ihtfp.com
       Computer and Internet Security Consultant
```

C.4.3. Example: Threshold ElGamal using Secret Sharing

During our work we faced some issues implementing Shamir's Secret Sharing in combination with ElGamal encryption and decryption. Each implementation itself seemed to work separately, both combined failed. For debugging purposes, we created an example using small numbers. Since the example turned out to be correct, we knew our problem had to be in our code. After peer reviewing the code using this example, we figured out, we had used the wrong modulus when combining the partial keys. The operation happens in $\mathbb{G}_q$ and not in $\mathbb{Z}_p$.

ElGamal Encryption

$$
\begin{aligned}
p = 23 \quad q = 11 \quad g = 2 \quad sk = 6 \in \mathbb{Z}_q \quad pk = 18 \in \mathbb{G}_q \\
\mathbb{Z}_q = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \\
\mathbb{G}_q = \{2, 4, 8, 16, 9, 19, 13, 3, 6, 12, 1\} \\
m = 12 \in \mathbb{G}_q \quad r = 7 \in_R \mathbb{Z}_q
\end{aligned}
$$

$$(c_1, c_2) \equiv (g^r, m * pk^r) \equiv (2^7, 12 * 18^7) \equiv (13, 3) \quad (mod \quad 23) \in \mathbb{G}_q \times \mathbb{G}_q$$

Create Shares

$$
\begin{aligned}
t = 3 \quad 1 \leq t \leq n \\
n = 5 \quad t \leq n \leq q \\
a_0 := x \quad a_1 = 2 \quad a_2 = 3 \quad a_i \in \mathbb{Z}_q \\
P(\alpha) := 3\alpha^2 + 2\alpha + x \in \mathbb{Z}_q[x]
\end{aligned}
$$

Shares:

$$
\begin{aligned}
Share = s_i = &\quad (i, s) \quad i \in \{1, ..., n\} \\
s_1 = &\quad (1, P(1)) = (1, 0) \\
s_2 = &\quad (2, P(2)) = (2, 0) \\
s_3 = &\quad (3, P(3)) = (3, 6) \\
s_4 = &\quad (4, P(4)) = (4, 7) \\
s_5 = &\quad (5, P(5)) = (5, 3)
\end{aligned}
$$

Partial Decryption

Incoming Shares $\mathbb{S} = \{s_i, ..., s_j\}$
$\mathbb{S} = \{s_3, s_4, s_5\}$
Remember: $c_1 = 13$

$$
\begin{aligned}
c_i' = (v_i, w_i) \in \mathbb{Z}_q \times \mathbb{G}_q \\
d(s_i, c_1) := (i, c_1^s) \\
c_1' = d(s_3, c_1) = (3, 13^6) \equiv (3, 6) \quad (mod\ 23) \\
c_2' = d(s_4, c_1) = (4, 13^7) \equiv (4, 9) \quad (mod\ 23) \\
c_3' = d(s_5, c_1) = (5, 13^3) \equiv (5, 12) \quad (mod\ 23) \\
\mathbb{W} = \{c_1', c_2', c_3'\}
\end{aligned}
$$

Lagrange Coefficients and Combine

Lagrange coefficients

$$\lambda_i = \prod_{j=1;j\neq i}^{t} \frac{-v_j}{v_i - v_j} \in \mathbb{Z}_q$$

$$v_i = 3 \quad v_j \in \{4, 5\}$$

$$\lambda_1 = \frac{-4}{3-4} * \frac{-5}{3-5} \equiv \frac{7}{10} * \frac{6}{9} \equiv (7 * 10^{-1})(6 * 9^{-1}) \equiv (7 * 10)(6 * 5) \equiv 10 \pmod{11}$$

$$v_i = 4 \quad v_j \in \{3, 5\}$$

$$\lambda_2 = \frac{-3}{4-3} * \frac{-5}{4-5} \equiv \frac{8}{1} * \frac{6}{10} \equiv (8 * 1^{-1})(6 * 10^{-1}) \equiv (8 * 1)(6 * 10) \equiv 7 \pmod{11}$$

$$v_i = 5 \quad v_j \in \{3, 4\}$$

$$\lambda_3 = \frac{-3}{5-3} * \frac{-4}{5-4} \equiv \frac{8}{2} * \frac{7}{1} \equiv (8 * 2^{-1})(7 * 1^{-1}) \equiv (8 * 6)(7 * 1) \equiv 6 \pmod{11}$$

Combine

$$A = \prod_{\substack{w \in \mathbb{W} \\ j \in \{1,..,3\}}} w_j^{\lambda_j} \in \mathbb{G}_q$$

$$A = 6^{10} * 9^7 * 12^6 \equiv 6 \pmod{23}$$

Decryption Remember: $c_2 = 3$

$$m = \frac{c2}{A} \in \mathbb{G}_q$$

$$m = \frac{3}{3} \equiv 3 * 6^{-1} \equiv 3 * 4 \equiv 12 \pmod{23}$$

$$12 = 12$$

D. User Documentation

Here we provide a small documentation how to install our add-on and where to set the necessary options in order to use cryptographic functions. Variables you can customize are enclosed in angle brackets `<variable>`. Commands you have to type in in an input field or command-line are denoted in a different font like e.g. `command-example`.

D.1. Prerequisites

We expect you have installed the following tools on the operating system of your choice:

1. **git** Install a git[1] client of your choice.
2. **Mozilla Firefox** Install the latest version of Mozilla Firefox[2] (min. required version is 63.0). Do not use the ESR release of Firefox.

D.2. Installation

1. Clone the Firefox add-on repository to the `<path>` of your choice using git (`ssh://git@github.com:ellr/firefox_addon.git`).
2. Open Firefox.
3. Type `about:debugging` into the address bar.
4. Click the button "Load Temporary Add-on" and navigate to the `<path>` you chose as directory for the plugin.
5. Select the `manifest.json` file in the folder and click "open".
6. You should now have the add-on listed under "Temporary Extensions" and should see a new icon in your toolbar.

Figure D.1.: Add-on icon

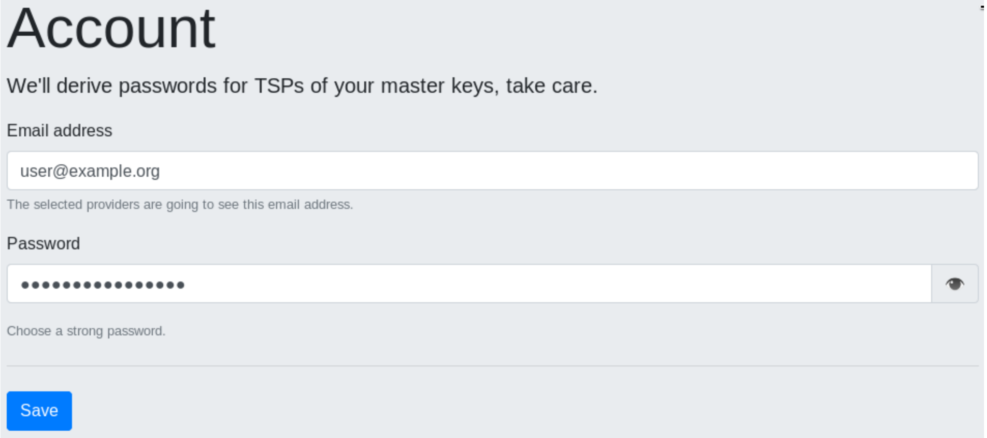
D.3. Add-on setup

1. Click on the add-on icon in your toolbar. A pane will open.
2. Click the gear icon in the right lower corner of the pane.
3. A new tab opens. Where you can click on "Account" on the left-hand side.

---

[1] `https://git-scm.com`
[2] `https://mozilla.org`

4. Enter your preferred email address and a secure password. By clicking on the eye on the right-hand side you can toggle the visibility of the password. Save the input afterwards.



Account

We'll derive passwords for TSPs of your master keys, take care.

Email address

user@example.org

The selected providers are going to see this email address.

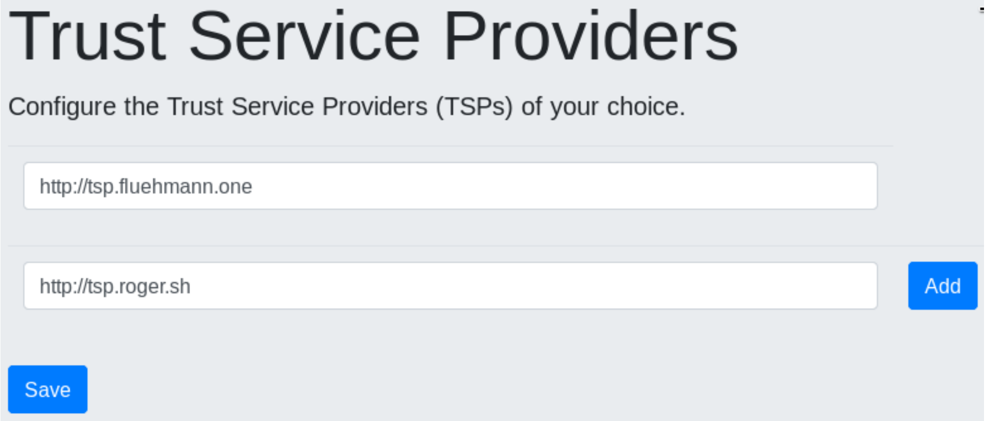
Password

••••••••••••••••

Choose a strong password.

Figure D.2.: Account

5. Select the next menu item "Providers" on the left-hand side.
6. Add the desired trust service provider URLs using the "Add" button after each entered TSP and save your input.



Trust Service Providers

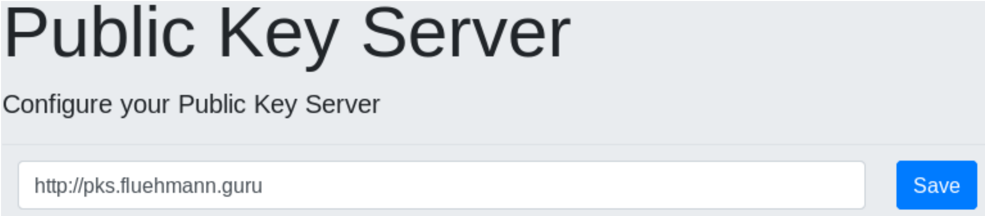
Configure the Trust Service Providers (TSPs) of your choice.

http://tsp.fluehmann.one

http://tsp.roger.sh

Figure D.3.: TSPs

7. On the same page you can specify the URL for the PKS. Save your input.



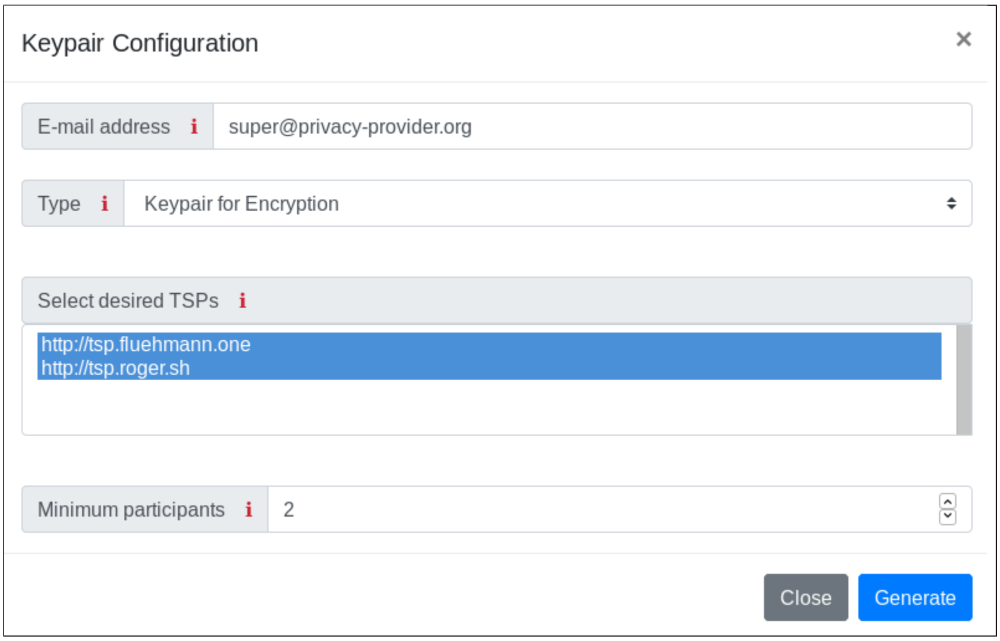
Public Key Server

Configure your Public Key Server

http://pks.fluehmann.guru

Figure D.4.: PKS

8. Now you can click the next menu-item on the left-hand side, "My Keys". Here we generate a new key pair by clicking on "generate new Keypair"
9. A new modal pops up. Select the desired parameters for your key.
   The email address belonging to the key, the purpose of the key (sign/verify or encryption/decryption), the desired TSPs and the minimal threshold.



Keypair Configuration

E-mail address: super@privacy-provider.org

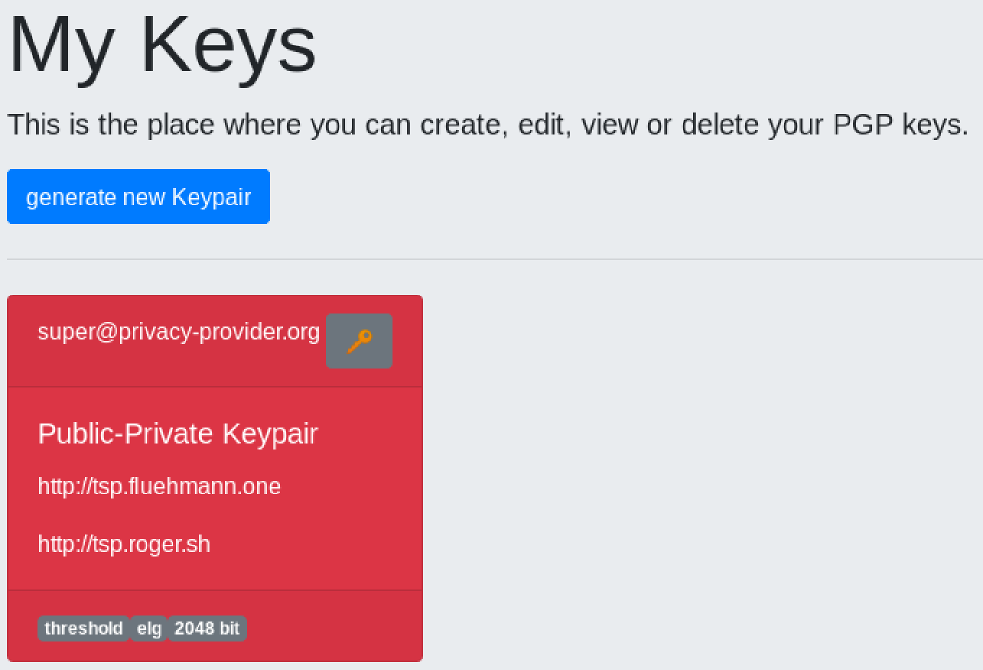
Type: Keypair for Encryption

Select desired TSPs:
http://tsp.fluehmann.one
http://tsp.roger.sh

Minimum participants: 2

Figure D.5.: Key Generation

10. Now you should have a new red card in the menu. The red card is a representation of a key pair you own. You can see the type of the key pair, and its properties like TSPs and key-size.



My Keys

This is the place where you can create, edit, view or delete your PGP keys.

super@privacy-provider.org

Public-Private Keypair

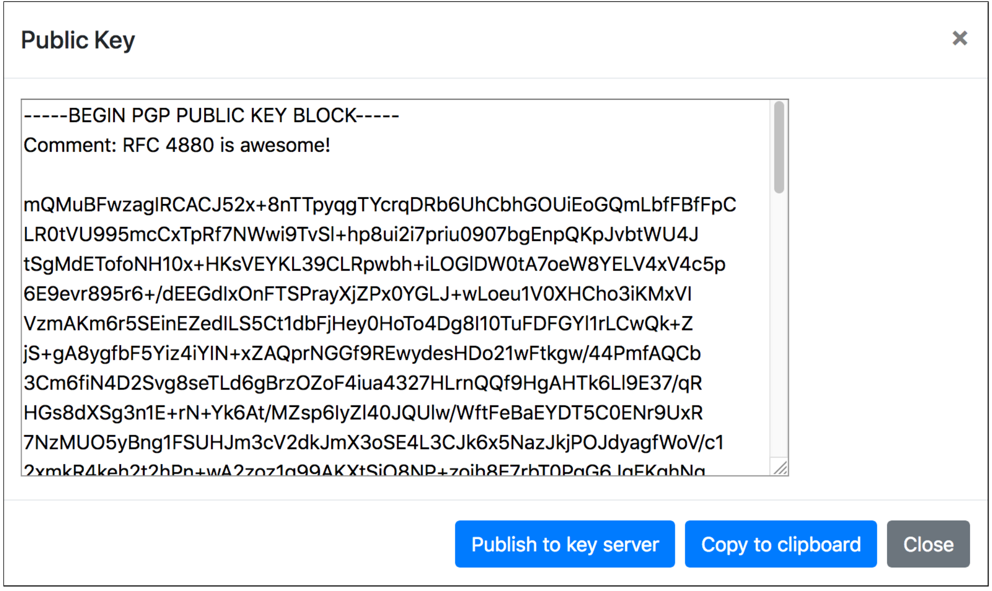
http://tsp.fluehmann.one

http://tsp.roger.sh

threshold elg 2048 bit

Figure D.6.: My Keys

11. By clicking on the key in the right upper corner on the red card, a new modal opens. Here you can publish the PGP public key of the key pair to the PKS or copy the PGP public key for another application using copy-paste. Other users will be able to use it afterwards and send you encrypted emails or verify your signatures. Publish your PGP public key now using "Publish to key server" and close the modal.



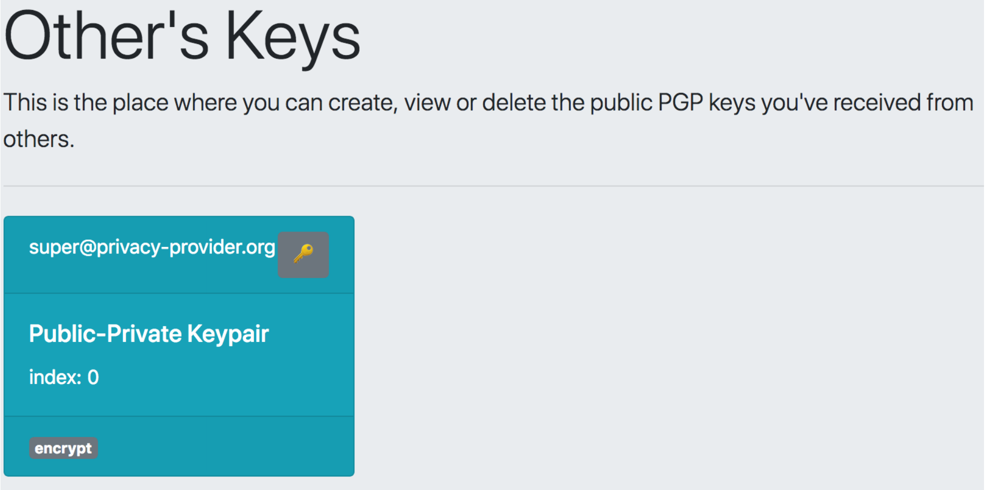
Public Key

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Comment: RFC 4880 is awesome!

mQMuBFwzaglRCACJ52x+8nTTpyqgTYcrqDRb6UhCbhGOUiEoGQmLbfFBfFpC
LR0tVU995mcCxTpRf7NWwi9TvSl+hp8ui2i7priu0907bgEnpQKpJvbtWU4J
tSgMdETofoNH10x+HKsVEYKL39CLRpwbh+iLOGlDW0tA7oeW8YELV4xV4c5p
6E9evr895r6+/dEEGdlxOnFTSPrayXjZPx0YGLJ+wLoeu1V0XHCho3iKMxVl
VzmAKm6r5SEinEZedlLS5Ct1dbFjHey0HoTo4Dg8l10TuFDFGYl1rLCwQk+Z
jS+gA8ygfbF5Yiz4iYlN+xZAQprNGGf9REwydesHDo21wFtkgw/44PmfAQCb
3Cm6fiN4D2Svg8seTLd6gBrzOZoF4iua4327HLrnQQf9HgAHTk6Ll9E37/qR
HGs8dXSg3n1E+rN+Yk6At/MZsp6lyZl40JQUlw/WftFeBaEYDT5C0ENr9UxR
7NzMUO5yBng1FSUHJm3cV2dkJmX3oSE4L3CJk6x5NazJkjPOJdyagfWoV/c1
```

Figure D.7.: Public Key

12. Navigate to the page displaying the public keys by clicking on the button "Other's Keys" on the left-hand side. Here you can see all PGP public key representations, including yours. They are denoted as turquoise cards. These cards have the small key in the right upper corner as well. Here you can only display and copy the PGP public key value but not publish it.



Other's Keys

This is the place where you can create, view or delete the public PGP keys you've received from others.

super@privacy-provider.org

Public-Private Keypair

index: 0

Figure D.8.: Other's Keys

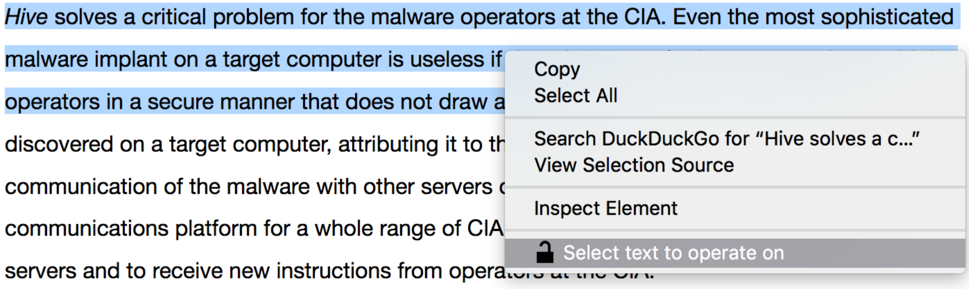
Note: In the "About" section you can delete all configurations of the add-on with the "Reset" button. The accounts on the web services will not be deleted in this case.

D.4. Cryptographic operations

In this section we presume having the add-on installed and configured as described in the sections Installation and Add-on setup. Additionally, you should have added two key pairs, one for signing and verifying and another for encryption and decryption. In the "Step Log" side pane on the left-hand side of your browser, the add-on informs you about the processes going on. Error of the environment and processes are marked in red or yellow. Informative notifications are displayed in green.

D.4.1. Encryption

1. Select the text you want to encrypt with your mouse and right click on it. Choose "Select text to operate on" in the context menu. If you put the cursor into a textarea or another editable field, you can select the html element as well using "Select element to operate on".

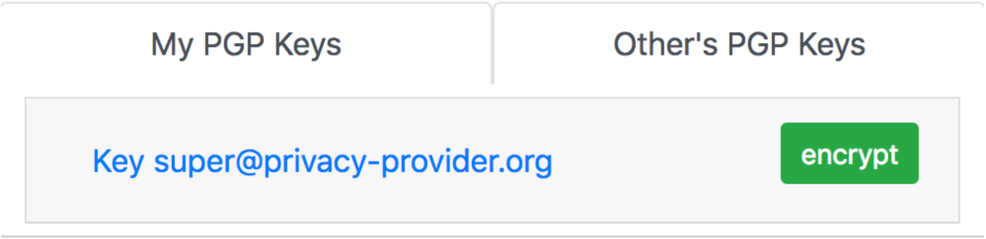


Hive solves a critical problem for the malware operators at the CIA. Even the most sophisticated
malware implant on a target computer is useless if
operators in a secure manner that does not draw a
discovered on a target computer, attributing it to th
communication of the malware with other servers c
communications platform for a whole range of CIA
servers and to receive new instructions from opera

Copy
Select All
Search DuckDuckGo for "Hive solves a c..."
View Selection Source
Inspect Element
Select text to operate on

Figure D.9.: Context menu

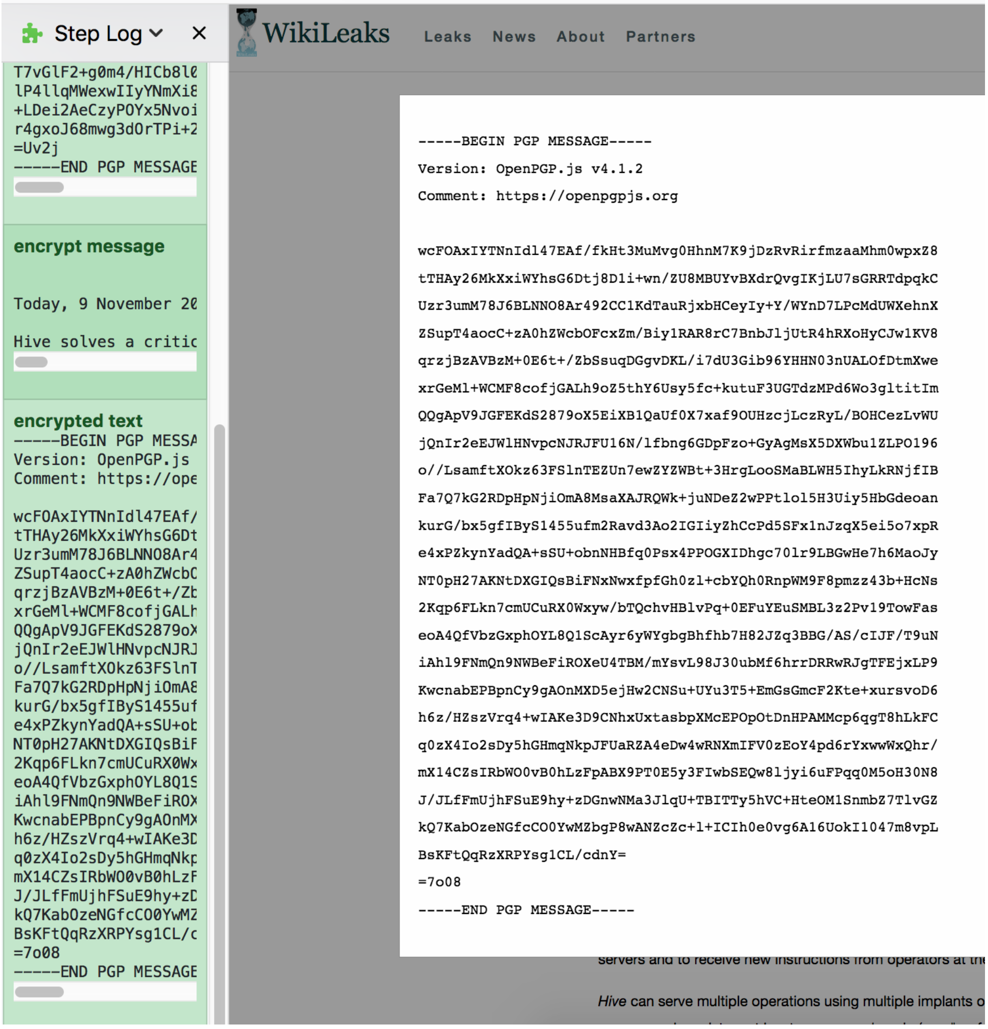
2. Open the add-on in the right upper corner in your Firefox. Select the tab "Other's PGP Keys". Now you have a couple PGP public key representations. Select the key you want to use for encrypting and click on "encrypt".



My PGP Keys
Other's PGP Keys
Key super@privacy-provider.org
encrypt

Figure D.10.: Other's PGP Keys

3. A new modal on the top of the webpage should appear now with the encrypted PGP message enclosed. You can select and copy the message and process it with your desired application (e.g. social media platform, web mail client).



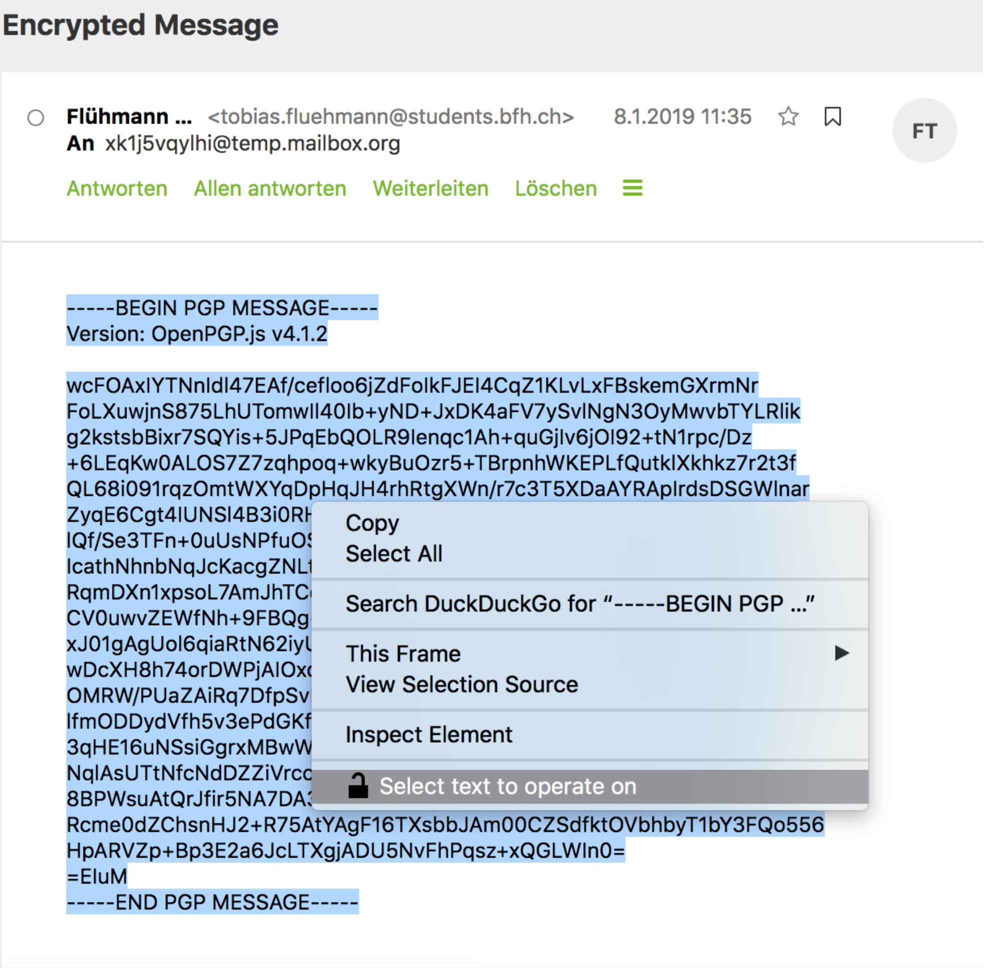
```
-----BEGIN PGP MESSAGE-----
Version: OpenPGP.js v4.1.2
Comment: https://openpgpjs.org

wcFOAxIYTNnIdl47EAf/fkHt3MuMvg0HhnM7K9jDzRvRirfmzaaMhm0wpxZ8
tTHAy26MkXxiWYhsG6Dtj8D1i+wn/ZU8MBUYvBXdrQvgIKjLU7sGRRTdpqkC
Uzr3umM78J6BLNNO8Ar492CC1KdTauRjxbHCeyIy+Y/WYnD7LPcMdUWXehnX
ZSupT4aocC+zA0hZWcbOFcxZm/Biy1RAR8rC7BnbJljUtR4hRXoHyCJw1KV8
qrzjBzAVBzM+0E6t+/ZbSsuqDGgvDKL/i7dU3Gib96YHHN03nUALOfDtmXwe
xrGeMl+WCMF8cofjGALh9oZ5thY6Usy5fc+kutuF3UGTdzMPd6Wo3gltitIm
QQgApV9JGFEKdS2879oX5EiXB1QaUf0X7xaf9OUHzcjLczRyL/BOHCezLvWU
jQnIr2eEJWlHNvpcNJRJFU16N/lfbng6GDpFzo+GyAgMsX5DXWbu1ZLPO196
o//LsamftXOkz63FSlnTEZUn7ewZYZWBt+3HrgLooSMaBLWH5IhyLkRNjfIB
Fa7Q7kG2RDpHpNjiOmA8MsaXAJRQWk+juNDeZ2wPPtlol5H3Uiy5HbGdeoan
kurG/bx5gfIByS1455ufm2Ravd3Ao2IGIiyZhCcPd5SFx1nJzqX5ei5o7xpR
e4xPZkynYadQA+sSU+obnNHBfq0Psx4PPOGXIDhgc70lr9LBGwHe7h6MaoJy
NT0pH27AKNtDXGIQsBiFNxNwxfpfGh0zl+cbYQh0RnpWM9F8pmzz43b+HcNs
2Kqp6FLkn7cmUCuRX0Wxyw/bTQchvHBlvPq+0EFuYEuSMBL3z2Pv19TowFas
eoA4QfVbzGxphOYL8Q1ScAyr6yWYgbgBhfhb7H82JZq3BBG/AS/cIJF/T9uN
iAhl9FNmQn9NWBeFiROXeU4TBM/mYsvL98J30ubMf6hrrDRRwRJgTFEjxLP9
KwcnabEPBpnCy9gAOnMXD5ejHw2CNSu+UYu3T5+EmGsGmcF2Kte+xursvoD6
h6z/HZszVrq4+wIAKe3D9CNhxUxtasbpXMcEPOpOtDnHPAMMcp6qgT8hLkFC
q0zX4Io2sDy5hGHmqNkpJFUaRZA4eDw4wRNXmIFV0zEoY4pd6rYxwwWxQhr/
mX14CZsIRbWO0vB0hLzFpABX9PT0E5y3FIwbSEQw8ljyi6uFPqq0M5oH30N8
J/JLfFmUjhFSuE9hy+zDGnwNMa3JlqU+TBITTy5hVC+HteOM1SnmbZ7TlvGZ
kQ7KabOzeNGfcCO0YwMZbgP8wANZcZc+l+ICIh0e0vg6A16UokI1047m8vpL
BsKFtQqRzXRPYsg1CL/cdnY=
=7o08
-----END PGP MESSAGE-----
```

Figure D.11.: Encrypted Text

D.4.2. Decryption

1. The decryption process is implemented in the same way as encryption. Select the message you want to decrypt and either press "Select text to operate on" or "Select element to operate on" in the context menu after selecting and right-clicking the text.



Encrypted Message

Flühmann ... <tobias.fluehmann@students.bfh.ch> 8.1.2019 11:35
An xk1j5vqylhi@temp.mailbox.org

FT
Antworten
Allen antworten
Weiterleiten
Löschen

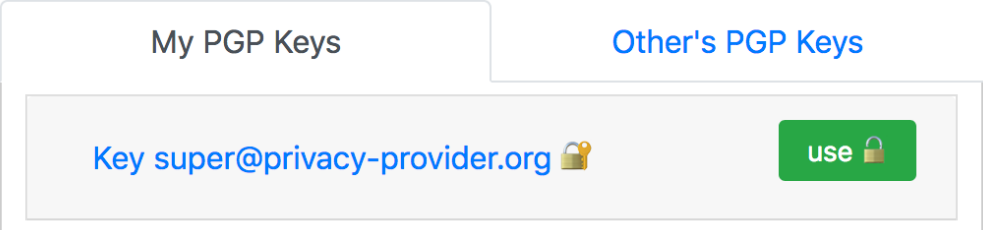
```
-----BEGIN PGP MESSAGE-----
Version: OpenPGP.js v4.1.2

wcFOAxIYTNnldl47EAf/cefloo6jZdFolkFJEl4CqZ1KLvLxFBskemGXrmNr
FoLXuwjnS875LhUTomwlI40lb+yND+JxDK4aFV7ySvlNgN3OyMwvbTYLRlik
g2kstsbBixr7SQYis+5JPqEbQOLR9lenqc1Ah+quGjlv6jOl92+tN1rpc/Dz
+6LEqKw0ALOS7Z7zqhpoq+wkyBuOzr5+TBrpnhWKEPLfQutklXkhkz7r2t3f
QL68i091rqzOmtWXYqDpHqJH4rhRtgXWn/r7c3T5XDaAYRAplrdsDSGWlnar
ZyqE6Cgt4lUNSl4B3i0RH
lQf/Se3TFn+0uUsNPfuOS
IcathNhnbNqJcKacgZNLt
RqmDXn1xpsoL7AmJhTC
CV0uwvZEWfNh+9FBQg
xJ01gAgUol6qiaRtN62iyU
wDcXH8h74orDWPjAlOx
OMRW/PUaZAiRq7DfpSv
lfmODDydVfh5v3ePdGKf
3qHE16uNSsiGgrxMBwW
NqlAsUTtNfcNdDZZiVrcc
8BPWsuAtQrJfir5NA7DA
Rcme0dZChsnHJ2+R75AtYAgF16TXsbbJAm00CZSdfktOVbhbyT1bY3FQo556
HpARVZp+Bp3E2a6JcLTXgjADU5NvFhPqsz+xQGLWIn0=
=EluM
-----END PGP MESSAGE-----
```

Copy
Select All
Search DuckDuckGo for "-----BEGIN PGP ..."
This Frame ▶
View Selection Source
Inspect Element
Select text to operate on

Figure D.12.: Context menu

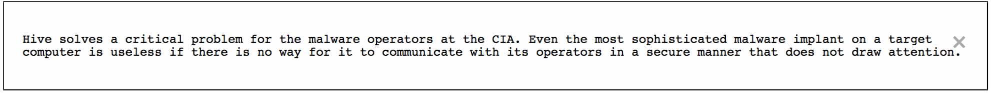
2. Open the add-on in the right upper corner in your Firefox. Select the tab "My PGP Keys". Now you have a couple PGP private key representations. Select the key you set up before for decryption and click on "use".



My PGP Keys
Other's PGP Keys
Key super@privacy-provider.org
use

Figure D.13.: My Keys

3. A new modal on the top of the webpage should appear now with the decrypted message.



Hive solves a critical problem for the malware operators at the CIA. Even the most sophisticated malware implant on a target computer is useless if there is no way for it to communicate with its operators in a secure manner that does not draw attention.

Figure D.14.: Decrypted Message

D.4.3. Signing

1. Select your text or your html element to operate on, as already visualized in figure D.9 and figure D.12.
2. Open the add-on in the right upper corner in your Firefox. Select the tab "My PGP Keys". Now you have a couple PGP private key representations. Select the key you set up before for signing and click on "use".

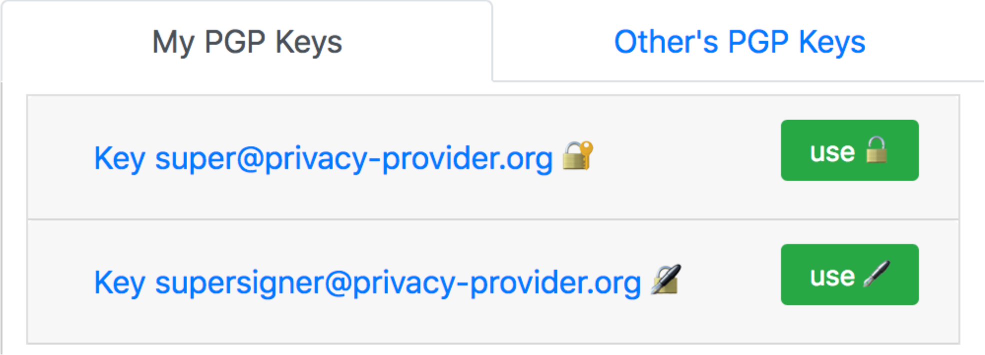
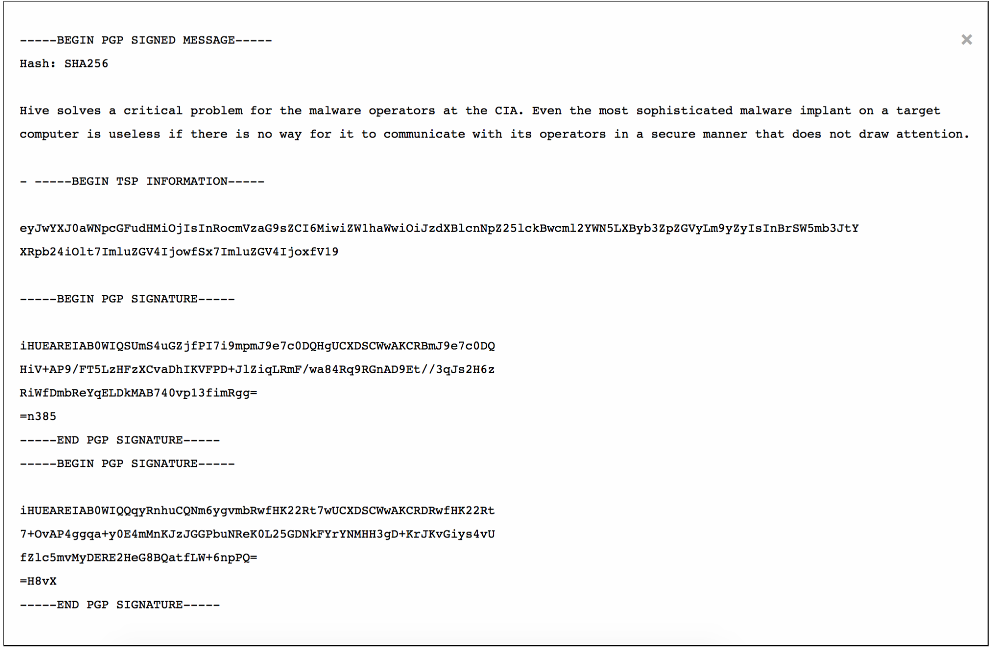

My PGP Keys
Other's PGP Keys
Key super@privacy-provider.org
use
Key supersigner@privacy-provider.org
use

Figure D.15.: My Keys

3. A new modal on the top of the webpage should appear now with the signed message.



```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA256

Hive solves a critical problem for the malware operators at the CIA. Even the most sophisticated malware implant on a target
computer is useless if there is no way for it to communicate with its operators in a secure manner that does not draw attention.

- -----BEGIN TSP INFORMATION-----

eyJwYXJ0aWNpcGFudHMiOjIsInRocmVzaG9sZCI6MiwiZW1haWwiOiJzdXBlcnNpZ25lckBwcml2YWN5LXByb3ZpZGVyLm9yZyIsInBrSW5mb3JtY
XRpb24iOlt7ImluZGV4IjowfSx7ImluZGV4IjoxfV19

-----BEGIN PGP SIGNATURE-----

iHUEAREIAB0WIQSUmS4uGZjfPI7i9mpmJ9e7c0DQHgUCXDSCWwAKCRBmJ9e7c0DQ
HiV+AP9/FT5LzHFzXCvaDhIKVFPD+JlZiqLRmF/wa84Rq9RGnAD9Et//3qJs2H6z
RiWfDmbReYqELDkMAB740vpl3fimRgg=
=n385
-----END PGP SIGNATURE-----
-----BEGIN PGP SIGNATURE-----

iHUEAREIAB0WIQQqyRnhuCQNm6ygvmbRwfHK22Rt7wUCXDSCWwAKCRDRwfHK22Rt
7+OvAP4ggqa+y0E4mMnKJzJGGPbuNReK0L25GDNkFYrYNMHH3gD+KrJKvGiys4vU
fZlc5mvMyDERE2HeG8BQatfLW+6npPQ=
=H8vX
-----END PGP SIGNATURE-----
```

Figure D.16.: Decrypted Message

D.4.4. Verifying

1. Select your text or your html element to operate on, as already visualized in figure D.9 and figure D.12.
2. Open the add-on in the right upper corner in your Firefox. Select the tab "Other's PGP Keys". Now you have a couple PGP public key representations. Select a key you set up before for verifying and click on "verify".

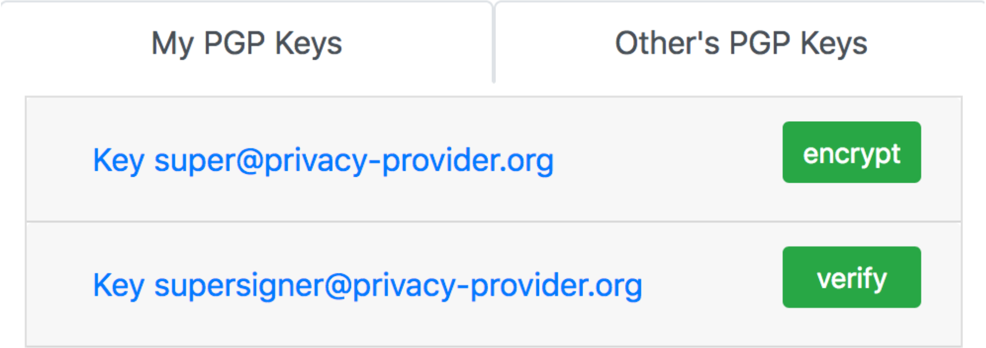
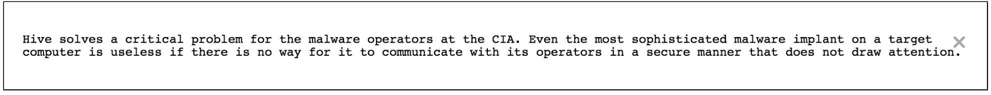

My PGP Keys
Other's PGP Keys
Key super@privacy-provider.org
encrypt
Key supersigner@privacy-provider.org
verify

Figure D.17.: Verification Keys

3. A new modal on the top of the webpage should appear now with the verified message. The Step Log on the left-hand side should stay green. Yellow or red messages show unsuccessful operations.



```
Hive solves a critical problem for the malware operators at the CIA. Even the most sophisticated malware implant on a target
computer is useless if there is no way for it to communicate with its operators in a secure manner that does not draw attention.
```

Figure D.18.: Decrypted Message

E. Content of CD-ROM

The content of the enclosed CD-ROM is separated in two main parts: documentation and source code.

Documentation

The directory `doc` contains:

- `code_documentation`
    - `firefox_addon`: HTML Code documentation of the add-on
    - `pks`: API documentation and test coverage of the PKS back-end
    - `tsp`: API documentation, test coverage, and HTML code documentation of the TSP back-end
- `meeting_protocols`: Meeting memos as markdown files
- `miscellaneous`
    - `book`: PDF published on `book.bfh.ch`
    - `final_day_exhibition_poster`: Poster for exhibition on final day
    - `final_day_presentation`: Slides of short presentation
    - `mockups`: UI mockups of add-on
- `project_management`
    - `project_schedule`: Estimated and effective project schedule
    - `requirements`: List of requirements
    - `screenshots_of_project_board`: Screenshots of GitHub project board
    - `time_sheet`: The file "journal.md" we have used to track our hours

Source code

The directory `src` contains:

- `firefox_addon`: Source code of the Firefox add-on
- `pks`: Source code of the PKS back-end
- `tsp`: Source code of the TSP back-end