Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Transparency and Security for Client-Side Encrypting Cloud Storage Applications

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von: Erik Nellessen
geboren am: 02.08.1989
geboren in: Meerbusch

Gutachter/innen: Prof. Dr. rer. nat. Jens-Peter Redlich
Prof. Dr. Ernst-Günter Giessmann

eingereicht am: ..... verteidigt am: .....

Contents

1 Introduction

Cloud storage applications are wide-spread in these days. They provide availability and reliability for their users' data.

In the status quo, most users do not use client-side encrypting cloud storage applications. That means that their files are transferred to the cloud storage as plaintext. Because of that, cloud storage providers are able to read those files. The privacy of the cloud storage users cannot be guaranteed.

At the same time, we witness extensive surveillance of digital communication by secret services (see, for example, [1]). Even if the cloud storage providers respect their users' privacy, secret services can make them release their users' data.

This is why we have to do client-side encryption to protect our privacy. That means that the users' files are encrypted before they are transferred to the cloud storage. We discuss what has to be considered when designing a client-side encrypted cloud storage system.

While doing this, we also focus on usability. This includes integrating our solution transparently. That means that users can still use the cloud storage application they are familiar with. They do not have to get used to another application. Many users use proprietary software to access their cloud storage. We show how it can be confined in order to prevent it from harming the users' privacy.

Lastly, we add a token to our system. A token is a small device, e.g. a smartcard or a smartphone. It contains a secret needed for the decryption of the user's files. The user can use the token to decrypt the cloud storage files on any PC system.

A token can increase availability of the data and add security and usability to the system. We point out what has to be considered to provide a significant gain in security when integrating the token into the system. At the same time, we describe ways to provide usability in our solution.

Ultimately, we describe measures to help people to enjoy the comfort of cloud storages while preserving their privacy.

1.1 Motivation

As we already described, client-side encryption is a requirement to provide privacy for cloud storage users. We based this statement on spying cloud storage providers and secret services. But even if the cloud storage providers were trustworthy and secret services respected their citizens' privacy, client-side encryption still is a requirement, as we show in this chapter.

We give an overview of the security issues of Dropbox[1] presented in [2].

To simplify describing attacks, we use "Alice", "Bob" and "Mallet" as placeholder names. Alice and Bob play the role of valid users. Mallet plays the role of an attacker.

Dropbox used a mechanism for data deduplication which used hash values. Whenever Alice added a file to her Dropbox, the client software split the file in chunks of 4MB.

[1] `https://www.dropbox.com/`, all hyperlinks have been accessed on May 25, 2016.

It then calculated the hash values of all the chunks and sent the hash values to the server. The server checked, for which of the hash values it already had the chunks. It then requested to upload only those chunks, which it did not have yet.

This mechanism can be used to download files from other users without having the permission to do so. The requirement to get Alice's file is to have the hash values of the chunks. Once Mallet has them, he pretends to upload Alice's file by sending the hash values to the Dropbox server. Afterwards, he requests "his" file from the server. The server will then send him Alice's file. See figure 1 for a graphical presentation of the attack.

For this attack, Mallet needs the hash values of the chunks of the files. It is unlikely for him to have these. He would have to somehow steal them before. Anyway, this attack could easily have been prevented by the Dropbox server.

Mulazzani et al.[2] also show how this and other weaknesses in the Dropbox protocol were exploited for illegal, hardly traceable file sharing. This was a real problem for Dropbox and not an accepted security weakness. Meanwhile, Dropbox fixed the problems described here. This shows that Dropbox made a mistake implementing their software. People should not rely on their cloud storage providers to implement a system without security problems. They should be able to control their privacy. This work is about helping people to do so.

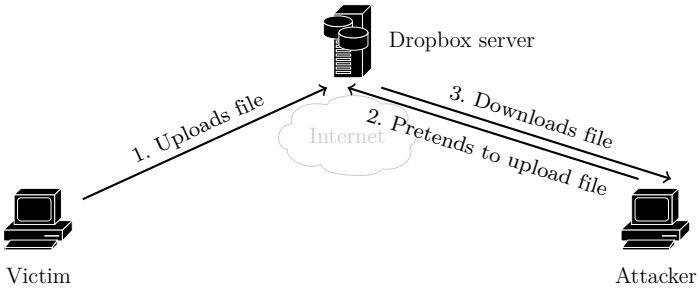

Figure 1: Getting another user's file without permission.

1.2 Problems addressed

There are three security levels at which a cloud-storage protocol might work: not encrypted, connection-encrypted or client-side encrypted. Figure 2 shows where Alice's data is visible in each method.

Methods one and two cannot provide privacy for the users. As the cloud storage provider receives the users' files in plaintext, the provider has access to them.

From an abstract point of view, client-side encrypted cloud storage systems can be represented as in figure 3.

In the first step, we concentrate on the "Certificate Management" part of cloud storage systems. We show security weaknesses in the design of current cloud storage

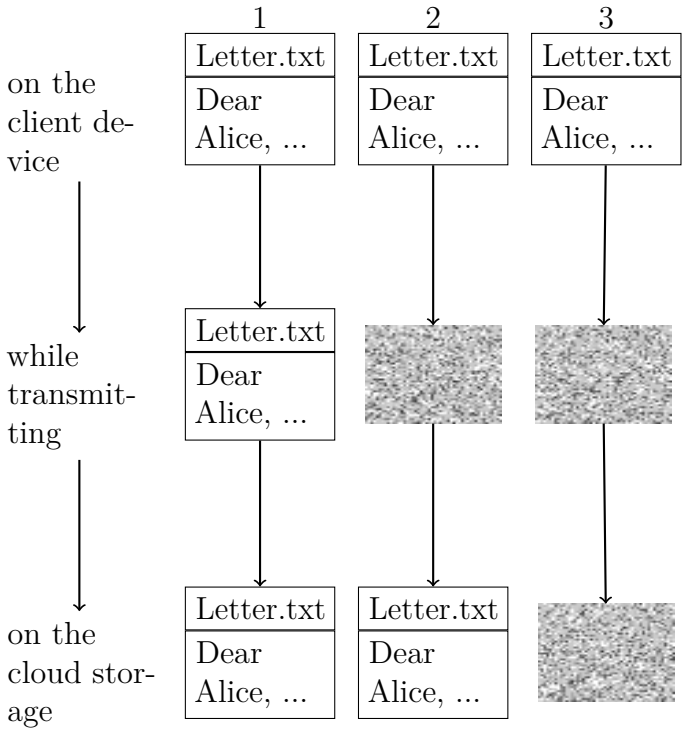

Figure 2: Idea for image from `https://www.wuala.com/img/how-it-works4.png` on `https://www.wuala.com/en/learn/technology`. As Wuala is out of service since November 15, 2015, the links do not work anymore. To point out that the illustration is not originally our idea, we name them anyhow.

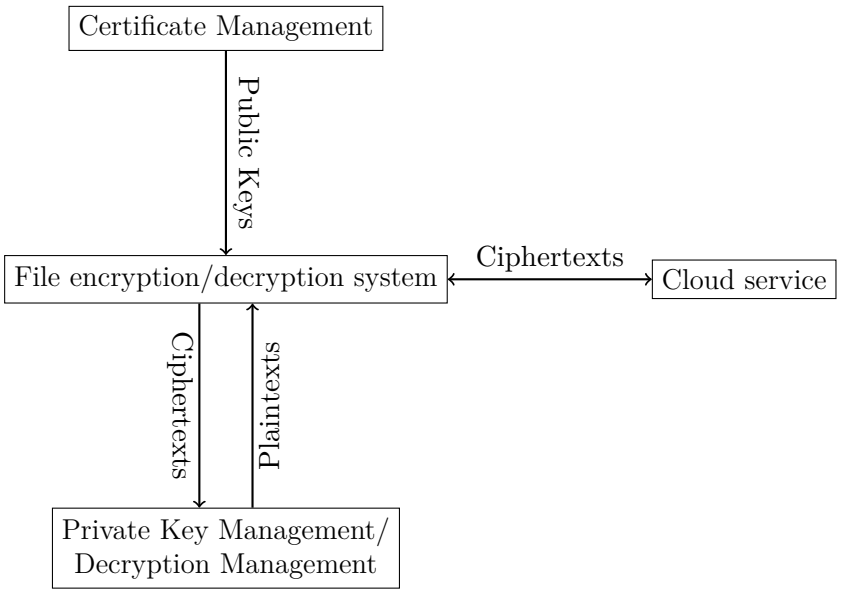

Figure 3: Abstract representation of client-side encrypted cloud storage systems.

applications. They require the user to trust in the cloud storage provider. We show security problems that need to be solved when designing a cloud storage system. The trade-off between comfort and security is discussed. We also provide an implementation of a system without the need of trust in the provider.

In the second step, we concentrate on the "Private Key Management/Decryption Management" part of cloud storage systems. We enable the user to control the decryption process. To do so, the private key resides on a token and the decryption is done in transactions, which need to be authorized by the user. We show requirements in the "File encryption/decryption system" needed for this approach (i.e. what we have to consider in step one). We then show how the file's meta data can be displayed on the display of a trusted device before the decryption of that file. The meta data is bound to the data which is decrypted, i.e. Alice sees authentically which one of her files will be decrypted.

1.3 Contribution

There is a lot of academic work about encrypted cloud storages. Anyhow, these publications mostly are about finding an efficient way to encrypt the files while still being able to share them.

In this work, we see this only as one module of the whole system (which we called "File encryption/decryption system" in figure 3). So the contribution in the first step is to discuss security problems on a more abstract level than it has been done until now. We analyze how public key verification can be done comfortably. We show scenarios that only make use of comfortable public key verification, i.e using smartphones and Quick Response (QR) codes. We also provide a protocol to verify public keys comfortably when only one device has a camera. We additionally analyze how a secure solution can also be transparent and show general limitations. Our proof of concept implements the described transparency and security measures.

In the second step, a token is added to the system. We analyze in what way security can be gained in this setup depending on the properties of the token. We describe a system that provides control over the decryption operations for the user. This is done by binding meta data to decryption keys. We also show general security limitations in this setup. Our contributions are the analysis of the possible security gains depending on the token, the procedure of binding meta data to decryption keys and our open source implementation of the described procedures.

2 Cloud storage without trusted third parties

2.1 Introduction

Motivation

As we have seen in section 1, only client-side encrypted cloud storages can be able to preserve their users' privacy. But even in client-side encrypted cloud storages, there are problems left to be solved.

When Alice wants to share files with Bob, she needs Bob's public key. The key may just be transmitted via the internet. But how does Alice know that it is *really* the public key of Bob, i.e. only Bob has the corresponding private key? If only Bob has the corresponding private key, we call the public key authentic.

The provider might solve the problem of authenticity of public keys by acting as a Certificate Authority. This means that the provider signs the public keys. Getting Bob's public key in an authentic way then works as shown in figure 4. After receiving the key, Alice checks if it (respectively its certificate) is signed with the provider's private key. If it is, Alice encrypts the secret she wants to share with Bob using Bob's public key. She then shares her secret with Bob as shown in figure 5. Bob can then download the encrypted secret as shown in figure 6 and decrypt it with his private key.

In this scenario, the cloud storage provider is able to perform a man-in-the-middle-attack. In the first step, the provider might not provide Bob's public key, but any other public key to which the provider knows the corresponding private key (as shown in figure 7). We denote Bob's public key (K_B^p) and the provider's public key (K_P^p). Alice checks if the certificate containing K_P^p contains a correct signature of the cloud storage provider. And it does, of course, as the cloud storage provider signed it (claiming it belongs to Bob). In the second step, Alice encrypts the secret she wants to share with Bob using K_P^p, as shown in figure 8. The provider decrypts the secret. The provider now knows the secret, which means that Alice's privacy is lost. But the provider may even play the man-in-the-middle-attack to the end, so nobody notices. It encrypts the decrypted secret using K_B^p. When Bob wants to download the encrypted secret, the provider provides the secret encrypted using K_B^p, as shown in figure 9. Bob decrypts it using his private key. The attack can be uncovered if Alice gets Bob's public key in a second, effectively authentic way and compares it to the version she got from the provider. But this is not likely to happen, if the software does not make clear that this is necessary in order to ensure privacy or does not provide a comfortable way to do this.

So this system only provides privacy for its users as long as the provider can be trusted. Even if the cloud storage providers are not interested in harming their users' privacy, secret services may be interested in the users' data. So they might make the providers reveal the users' data. That is why it is not a good idea to have the privacy depend on the provider. For this reason, eliminating the role of the provider as a trusted third party is crucial.

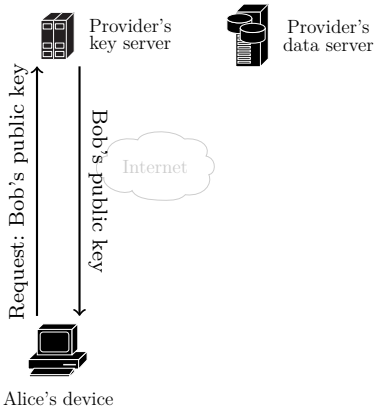

Figure 4: Alice gets Bob's public key.

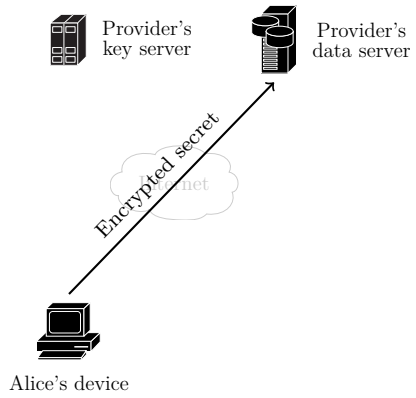

Figure 5: Alice uploads an encrypted secret for Bob.

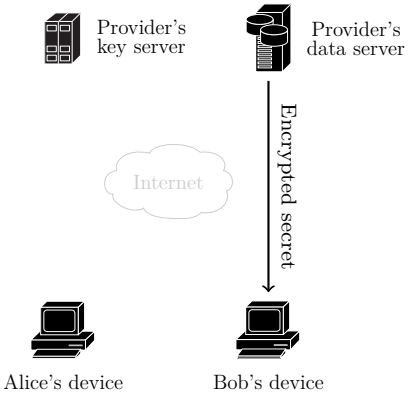

Figure 6: Bob downloads the encrypted secret and decrypts it on his PC.

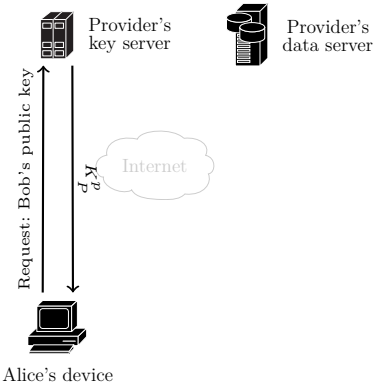

Figure 7: Alice requests Bob's public key, but the provider gives her another one.

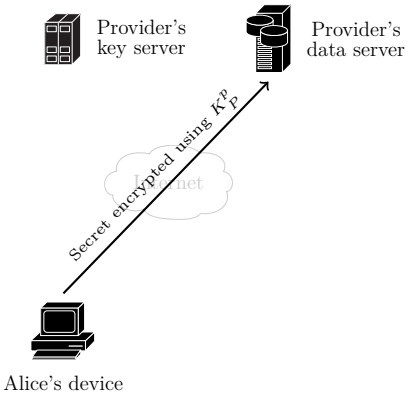

Figure 8: Alice uploads an encrypted secret for Bob.

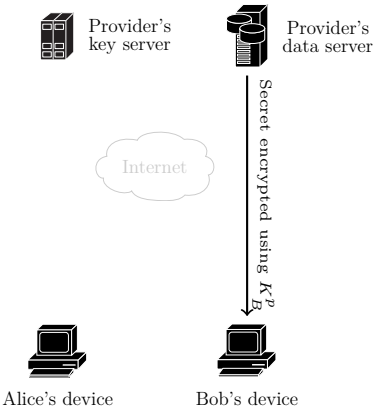

Figure 9: Bob downloads the encrypted secret and decrypts it on his PC.

Problems addressed in this part

We show existing solutions for the "File encryption/decryption system" and the "Certificate Management" component (from figure 3). We explain in what way they have to work together to form a client-side encrypted cloud storage. We also analyze how current implementations handle certificate management. Afterwards, we propose another solution in which users do not need to trust their providers.

In this solution, users do the public key verification instead of the providers. We show how this can be done in a comfortable way. Our solution is also transparent. This means that users can still use their original cloud storage application. We demonstrate how we can integrate the security measures transparently in the existing system.

The attacker in this part (i.e. section 2) has the ability to intercept and manipulate the data we use in our cloud storage. We assume that no attacker can manipulate our operating system or our hardware. This means, that our operating system and hardware can be trusted. What we can do in case this assumption is not warranted is described in the second part of this work, see section 3.

As this is an academic work, the system we present here is not supposed to be the most practical solution concerning cloud storages. It is supposed to depict in what way a system without a trusted third party can be built, and what limitations concerning usability exist. We do not expect cloud storage providers to adapt this system, as its usability is limited. Anyhow, we explain the limitations and suggest measures to improve usability in systems without trusted third parties.

Contribution

This work analyzes the security of client-side encrypted cloud storages not only on the level of the encryption, but in a more abstract way. This means that we show how the components of the system have to be combined in order to achieve security. While doing this, we also consider usability. We show how usability can be improved and at what points trade-offs between usability and security exist.

When doing public key verification, usability can be increased using smartphones and QR codes. We contribute the description of specific scenarios. In these scenarios, all required public key verifications for usual cloud storage use cases are done comfortably.

Doing a public key verification with exactly one device that does not include a camera does at first glance not seem to be possible in a comfortable way. This is because the public key verification on the device without camera cannot be done by scanning the QR code of the other device's public key. We contribute a protocol, which makes comfortable public key verification possible, anyhow.

When securing an existing cloud storage system, we can create a new program. Alice would then be supposed to use the new program instead of the original program. This approach is not the most comfortable one, as Alice has to get used to a new program. We contribute a system which integrates transparently. That means that Alice gains security, but can still use the original program.

Our implementation proves that the system as described in this part works in practice.

We do not include efficiency/scalability problems, as these are not important in our use case. We do not discuss consistency problems like in network file systems for the same reason.

2.2 Related work

In chapter 2.1, we demonstrated, why it is necessary to enable cloud storage users to client-side encrypt their files. Of course, academic work and practical implementations already exist in the field of client-side encrypted cloud storages. In this chapter, we integrate our work with existing publications.

To do so, we first have a look at articles about examinations of cloud storage providers. We will see that existing publications concluded that existing client-side encrypting cloud storages are prone to the man-in-the-middle attack we described in chapter 2.1.

We then examine existing programs for encrypting cloud storages. These programs use existing cloud storage providers and client-side encrypt the files for the user. We will see that these programs do not work transparently. They also do not meet important security requirements and are prone to the man-in-the-middle attack we described in chapter 2.1.

After that, we examine articles providing ideas for the "File encryption/decryption system"(see figure 3). We will see, that none of the examined publications includes a reusable open source implementation. Our work does not focus on the "File encryption/decryption system". This is why we just adopt ideas from the examined publications and create a simple "File encryption/decryption system" for our proof of concept implementation.

A lot of work is done in the field of homomorphic cryptography with the focus on client-side encrypted cloud storages. The idea is to encrypt the data, but still enable the cloud storage server to perform operations on the encrypted data (without harming the user's privacy). The examined publications are not directly important for our work. We just show, in what way this field is connected to our work.

From an abstract point of view, this part of the work is about securing an existing communication procedure. That is why we examine existing publications that already have treated this subject. We will find out that securing existing communication procedures is already done for communication via the Transmission Control Protocol (TCP)/Internet Protocol (IP) stack.

Articles about examinations of cloud storage providers

Duane C. Wilson and Giuseppe Ateniese [3] examined the cloud storage providers Wuala, Tresorit and Spider Oak in regard security issues. All of these services provide client-side encryption in their applications. Duane C. Wilson and Giuseppe Ateniese concluded that all three providers work as Certificate Authorities (CAs). That means that the confidentiality of Alice's data does not only depend on the confidentiality of Alice's private key. She needs to trust her provider. In chapter 2.1 we have shown

that this is not an acceptable trust model (as it enables the provider to perform a man-in-the-middle-attack).

Virvilis et al. [4] examined and compared existing architectures and infrastructures of secure cloud storages. This article gives a great overview of existing cloud storage providers and existing solutions for the "File encryption/decryption system" part. It shows that the examined cloud storage providers do not provide an acceptable security level. Virvilis et al. also define a set of criteria which can be used to compare solutions. Anyhow, this work does not add any criteria to ours (which we define in the segment "Programs for encrypting cloud storages"), as they are not important for our use case. The criteria do not concern public key management.

Besides the described attacks, there are still other ways to get a cloud storage user's data. Another possibility to attack a client-side encrypted storage is to attack the browser, given that a web interface for accessing the cloud is used [5, 6]. Whether that attack is possible in a client-side encrypted cloud storage depends on Alice's view while using the web interface. If she sees her files in plaintext, the attack is possible. If she sees only the encrypted files, the attack is not possible.

Programs for encrypting cloud storages

A client software should, in our opinion, meet the following requirements to be able to ensure privacy and usability:

- Simple file sharing
- Easy file versioning
- Portability
- User acceptance
- Open source code
- Secure encryption/providing authenticity
- Security only relying on the private key of the client

It is obvious that not satisfying one of these requirements leads to insecurity or bad usability.

CryptoHeaven[2] is a provider which ensures privacy by providing an open source, client-side encrypted cloud storage service. However, users have to pay to use this service and would have to change their cloud storage provider. There also is no statement about the possibility of sharing files. Hence, user acceptance is probably not given for this provider in the use case we assume in this part of the work.

A promising approach is to use a software that does a client-side encryption like Cryptree, while still using an untrusted provider. Like this, users can still use the

[2] `http://www.cryptoheaven.com/offsite-backup/offsite-backup.htm`

software they are used to. There exist several implementations who took this approach. This approach cannot satisfy our requirement of open source code, of course, if the provider's client software is not open source. We could achieve an acceptable level of security by executing the client software in a sandbox environment. This way, we could make sure that the client software can only access files in the cloud folder.

One implementation, which works on top of existing cloud storage providers, is Boxcryptor[3]. Boxcryptor claims to provide secure cloud storage for all major cloud providers. For the following reasons, Boxcryptor does not meet our requirements:

- The software is not open source.

- When sharing data with Bob, we get Bob's public key from the Boxcryptor key server[4], see figure 10. There is no user-side checking of the authenticity of Bob's public key involved, so trusting the public key means trusting the Boxcryptor key server. Our attacker Mallet, who controls the Boxcryptor key server, could easily insert a public key from which he knows the private key. So the cloud provider and Mallet could work together to perform a man-in-the-middle-attack as described in chapter 2.1. Mallet could be member of the Boxcryptor company or someone who gained control over the Boxcryptor key server (secret services, crackers).

- Alice's private key is stored on the Boxcryptor server. It is encrypted with her password[5]. Like this, it is easy for her to get access to the encrypted files from another device. Alice logs in to the Boxcryptor server. To do so, she provides the user name and the (salted) hash value of the password (we denote this hash value $h(pw, salt)$). The Boxcryptor server sends the private key (which is encrypted using a key derived from the password) to the user. We denote this ciphertext $E_{pw}(K_A^s)$. It can then be decrypted on Alice's device using Alice's password. For a graphical presentation of the process, see figure 11.
  Storing private keys on devices not controlled by the user is generally a bad idea, even if the private keys are encrypted, because the security of the whole system relies on them. Most users choose a weak password to encrypt their private key, as strong passwords are hard to create and remember. Making users choose really secure passwords cuts down usability. Encrypted private keys can be decrypted using Brute-Force-Attacks. These attacks usually use password lists to get the password. Once Mallet has the encrypted private key, he can try to break the password in parallel and with unlimited tries. That is why the duration of the password breaking process is mostly dependent on the amount of money Mallet can spend for his (highly) parallel password breaking system. An attacker with a lot of money could for example be a secret service.

[3] `https://www.boxcryptor.com/en`
[4] `https://www.boxcryptor.com/en/technical-overview#anc03`
[5] `https://www.boxcryptor.com/en/technical-overview#anc09`

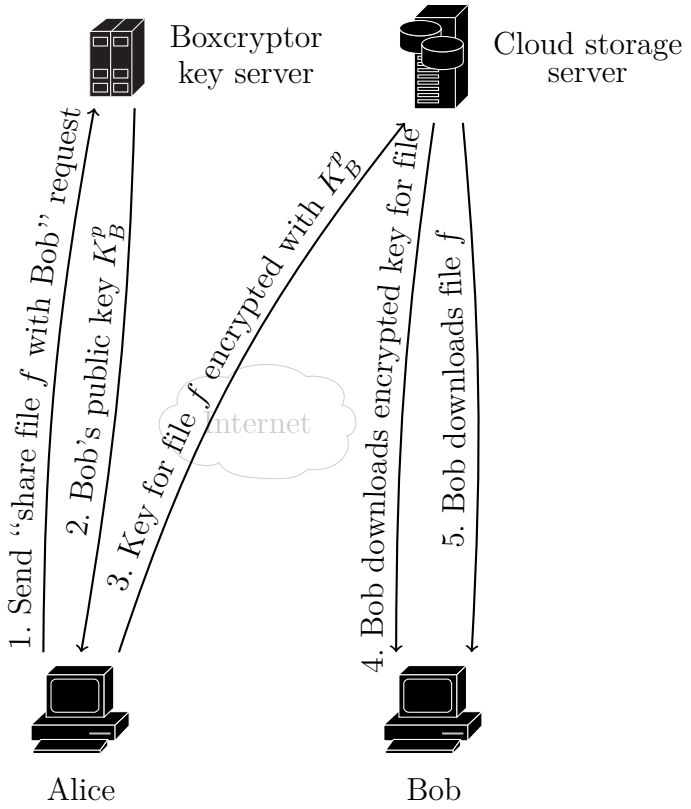

Figure 10: Alice shares a file with Bob. After step 5, Bob is able to decrypt the downloaded file f. This picture does not show how Alice grants Bob access to the file on the cloud storage server. This is probably done by the Boxcryptor client software.

Other providers, like Sookasa[6], do not even reveal how their systems work.

Another approach of providing confidentiality of data stored on a cloud storage is to add a trusted device that is responsible for the privacy of the users of the attached devices [7, 8]. This device is placed at the gateway of the network. It works similar to a proxy.

[6] `https://www.sookasa.com/how-it-works/`

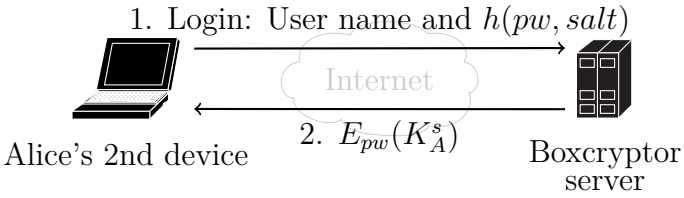

Figure 11: Alice adds a second device to her Boxcryptor account. After step 2, she can decrypt her private key. With the private key, she can decrypt the files in her cloud storage.

Articles providing ideas for the "File encryption/decryption system"

The "File encryption/decryption system" (first mentioned in figure 3) is a part of a client-side encrypting cloud storage system. It describes in what way Alice's files are encrypted. While providing encryption, it must also enable Alice to still share her files with Bob. Other criteria could be simplicity, usability, or efficiency.

There exist several papers on the issue of creating a good "File encryption/decryption system", e.g. Cryptree [9], Key to Cloud [10] (K2C), CloudSeal [11], Cloudasec [12]. In this work, we do not want to focus on developing new ideas for the "File encryption/decryption system". We would like to use an open source implementation of some previous work. The only paper we could find with an open source implementation is K2C.

Zarandioon et al. [10] propose a client-side encryption scheme called K2C, based on Attribute-Based Encryption (ABE) [10, page 3, first paragraph of section 2]. ABE [13] is a public-key encryption scheme. In this scheme, the ciphertext and the public and private key have attributes. Only private keys with the matching attributes can decrypt ciphertexts with specific attributes. The private keys are created and distributed by a trusted third party (Private Key Generator (PKG)). An advantage of this scheme is the possibility to add persons that should be able to decrypt after the encryption.

More concrete, the K2C scheme is based on Key-Policy Attribute-Based Encryption (KP-ABE) [14] [10, page 6, first paragraph of section 2.3]. To explain KP-ABE, we first explain Ciphertext-Policy Attribute-Based Encryption (CP-ABE) [15]. In CP-ABE, the ciphertext includes a boolean formula of attributes. To decrypt it, the attributes of the private key must evaluate the boolean formula to true. KP-ABE is just the other way around. The ciphertext has a set of attributes and the private key includes a boolean formula, which must evaluate to true for a successful decryption.

Part of the K2C publication is a library for Hierarchical Identity-Based Encryption (HIBE) [16] [10, page 3, last point in the itemization of chapter 1]. To understand, what HIBE is, we first explain Identity-Based Encryption (IBE). IBE [17] allows users to derive public keys from trivial information. This information could for example be a name or an e-mail address. As in ABE, a PKG is used to create and distribute the private keys. Actually, ABE is based on IBE and just regards a set of attributes as

an identity [13, page 1, in the abstract]. HIBE is an IBE with more than one PKG. There is one root PKG and sub-PKGs, which distribute private keys in their domain.

So again we need a trusted third party (as PKG). But we could make the owner of the files the PKG. Then, having the PKG would not contradict eliminating the trust in third parties. Alice would be the (root) PKG distributing private keys for decrypting her files. Distributing the private keys could be done via a confident and authentic channel. We could establish such a channel by doing a proper certificate management.

K2C provides three Java programs:

1. K2C Framework[7]
2. KP-ABE Library[8]
3. HIBE Library[9]

The K2C framework uses the cloud storage provider's Application Programming Interface (API). That is why the storage provider has to meet certain requirements. We have to write provider specific code. The quality of this software (i.e. all three Java programs) does not make a good impression. There is a lot of commented out code and no documentation. Also, there is just this one version on sourceforge. There is no visible development. That is why we decided not to use the K2C implementation for the "File encryption/decryption system" part of our client-side encrypted cloud storage application.

Grolimund et al. [9] proposed a file structure for a client-side encrypted cloud storage called Cryptree. The main goals of the system design was simplicity and usability. Cryptree has been implemented by Wuala[10] (a cloud storage provider). There is no open source implementation available. Anyhow, this paper provides some simple ideas for designing the "File encryption/decryption system" part.

Cryptree relies on folder structures. A folder contains information about itself (name, date of creation, etc.), about the files and folders it includes, and about the parent folder. The idea of the Cryptree is to encrypt all of this information and to use different keys for different information. The keys for the different decryptions are linked in a way that allows deriving all needed keys from only one key.

The links between keys are called cryptographic links. There are symmetric links and asymmetric ones. Symmetric links are created using symmetric encryption, for example AES, to encrypt a key. So to create a symmetric link from $\mathcal{K}_A$ to $\mathcal{K}_B$, we encrypt $\mathcal{K}_B$ with the AES procedure using $\mathcal{K}_A$ as the secret key. The result is the ciphertext $E_{\mathcal{K}_A}(\mathcal{K}_B)$, as shown in figure 12. Everyone who knows $\mathcal{K}_A$ and $E_{\mathcal{K}_A}(\mathcal{K}_B)$ can derive $\mathcal{K}_B$, as shown in figure 13. The advantage of symmetric links is performance. The disadvantage is that we have to create a shared secret between two communicating

[7] `https://sourceforge.net/projects/key2cloud/`
[8] `https://sourceforge.net/projects/kpabe/`
[9] `https://sourceforge.net/projects/hibe/`
[10] `https://www.wuala.com/en/learn/technology`, Wuala is out of service since November 15, 2015. That is why this reference does not work anymore.

parties prior to the encryption. Asymmetric links use asymmetric cryptography, for example RSA. Let K_A be an asymmetric key consisting of K_A^s as the private key and K_A^p as the public key. Let K_B be the key we want to establish a link to from K_A. We then encrypt K_B with the RSA procedure using K_A^p as the key. The result is the ciphertext $E_{K_A^p}(K_B)$, as shown in figure 14. Everyone who knows K_A^s and $E_{K_A^p}(K_B)$ can derive K_B, as shown in figure 15. The advantage of asymmetric links is the possibility of creating them without having to create a shared secret over an insecure channel before (assuming we received the public keys in an authentic way). The disadvantage is the bad performance.

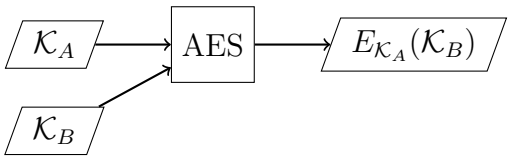

Figure 12: Creation of a symmetric link.

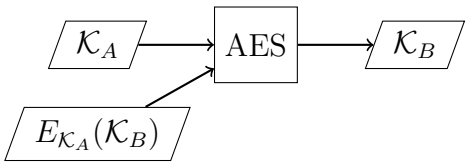

Figure 13: Deriving $\mathcal{K}_B$ using a symmetric link.

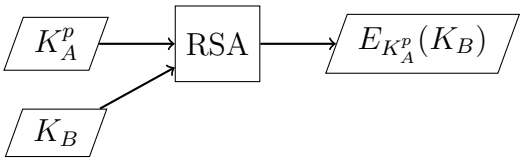

Figure 14: Creation of an asymmetric link.

A folder consists of the following data:

Representative data = Name, creation date...	Parent folders	Subfolders	Files

Cryptree's idea is to encrypt every kind of data with its own key. So to decrypt all the data associated with the folder, we need the following keys:

Data Key	Backlink Key	Subfolder Key	Files Key

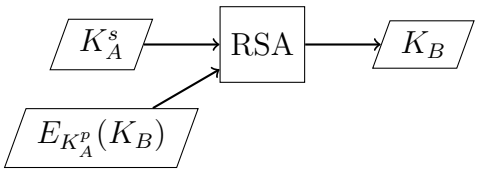

Figure 15: Deriving K_B using an asymmetric link.

With the data and backlink key, we can find out information about the folder and its parent folder. With the subfolder key and the files key, we can decrypt the subfolders and files.

We also make it possible to derive all keys for one folder from the subfolder key. Additionally, the backlink key from the parent folder can be derived from the backlink key of the current folder. So if we know the subfolder key of the folder *user* in the path */home/user/documents*, we get the structure as shown in figure 16.

To be able to share a folder with other users, we add one key. This key is called the clearance key. All other keys can be derived from the clearance key. It is enough to make it possible to derive the subfolder key, because we can derive all other keys from the subfolder key, see figure 17.

So to share a folder (that means the folder itself, all the files and subfolders included in the folder, and the information about parent folders) with Bob, we enable Bob to derive the clearance key from his private key and the encrypted clearance key. That means, we add an asymmetric link from his private key to the clearance key.

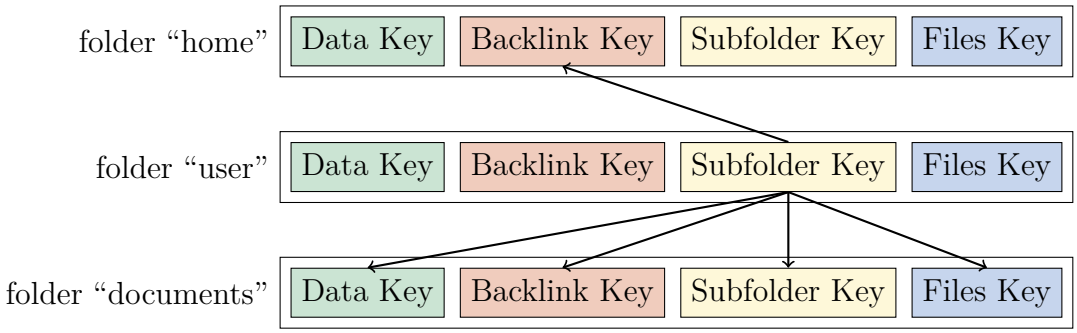

Figure 16: Derivable keys when knowing the subfolder key of the folder “user”. Keep in mind that all keys of the folder “user” can be derived from the subfolder key of the folder “user”.

Cryptree uses symmetric links whenever possible because they are more efficient. Asymmetric links are only used to derive the clearance keys from private keys. This enables us to share data with Bob without having to exchange a secret with him.

We do not explain Cryptree any further at this point. Of course, the Cryptree paper contains more ideas and more details than we presented here. But as already stated, we

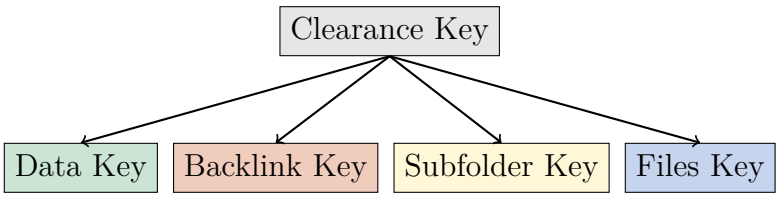

Figure 17: Deriving all keys from the clearance key.

do not concentrate on the "File encryption/decryption system" part. That is why we just adopt some of the Cryptree ideas to create a simple "File encryption/decryption system" part. We adopt the folder structure, i.e. all files in a folder are encrypted with one key. We also adopt the idea of being able to decrypt all subfolders of a folder (if we can decrypt the folder). We do so by creating a symmetric link from the parent folder key to the subfolder key. By following these ideas, our client-side encrypted cloud storage can be used in an intuitive way.

The project zkcrypto[11] claims to implement Cryptree. It does not have any public documentation. There has been no progress since October 2011. The Cryptree interface requires a login and a password, which does not match our understanding of the system. Our understanding is that the interface requires files and a public key. Cryptree/the "File encryption/decryption system" then encrypts those files and provides access for the owner of the corresponding private key. For these reasons, we did not use the zkcrypto implementation.

Cloudseal is another publication providing a "File encryption/decryption system". It also does not provide an open source implementation. It aims at high efficiency and scalability [11, page 9, chapter 5]. Of course, these are important goals when designing a "File encryption/decryption system". On the other hand, Cloudseal did not focus on simplicity and usability. But these goals are more important in our use case. We do not care that much about efficiency and scalability, as it is not that important in our use case. The number of persons we share our data with is usually smaller than 10. Also, in Cloudseal, the cloud service provider is required to be "semi-honest; that is, it follows the protocol and operations defined in CloudSeal, but it may actively attempt to gain knowledge of cleartext of the content" ([11, page 3, first paragraph of chapter 2]). That means, the provider has to behave in a certain way. That is not what we want, as we do not want to trust in the provider to be honest at all. For these reasons, we do not adopt the ideas from Cloudseal.

Another publication providing a "File encryption/decryption system" is Cloudasec. Cloudasec also does not provide an open source implementation. As K2C and Cloudseal, Cloudasec also does not state simplicity as a main requirement. Again, efficiency and flexible access control are important goals [12, page 2, chapter 2]. So just like Cloudseal, Cloudasec does not provide enough simplicity and usability for our use case. We decided not to include ideas from Cloudasec for that reason.

[11] `https://code.google.com/archive/p/zkcrypto/`

Articles about homomorphic cryptography

There are also other subjects of research in the area of client-side encrypted cloud storages. Performing operations on encrypted data, like search queries, without getting to know anything about the data (homomorphic encryption) is discussed in many publications, for example [18–21]. This is an important subject of research at the moment. Anyhow, it is not very important for our work, as it is only a detail of the "File encryption/decryption system". As already stated, we do not focus on the "File encryption/decryption system".

Securing existing communication procedures

An example for securing existing communication procedures is stunnel[12]. Stunnel can be used to secure communication done by programs which do not support confidentiality or authenticity on the Transport Layer Security (TLS) layer. From an abstract point of view, this is what Boxcryptor and others are trying to do. They also want to bring in security after the system design. We also take this approach, but consider our requirements (see the segment "Programs for encrypting cloud storages") while doing so.

Conclusions

As we have seen, existing client-side encrypting cloud storages are prone to the man-in-the-middle attack described in chapter 2.1. Programs for encrypting cloud storages also cannot prevent this attack. Additionally, they do not meet other security requirements, e.g. being open source.

These results motivate to create a system without the described disadvantages. In the following chapters, we analyze how our requirements can be fulfilled. We then present a system which has been designed considering our requirements.

As we have observed, no proper open source implementation is available for the "File encryption/decryption system". Anyhow, we can use existing ideas from Cryptree to implement a simple "File encryption/decryption system".

We do not address problems like consistency etc., like they are discussed in network file systems. We only treat the cloud storage use case in which users share photos, videos, documents etc. The network file system problems are not a big issue in that setting.

2.3 Design goals and abstract view

In this chapter, we abstractly describe our system design. The first two segments cover the establishment of secure channels in a comfortable way using smartphones and QR codes. We describe how devices are added and files are shared in our system. We show how all required public key verifications for these use cases can be done comfortably.

[12] `https://www.stunnel.org/index.html`

After that, we present a protocol to do a comfortable two-way public key verification, when only one device has a camera.

The latter segments are about transparently securing existing communication systems. First, we analyze why and how it is possible to secure a cloud storage transparently. To do so, we show how other communication procedures have been secured belatedly and compare these systems to a cloud storage. We then explain how transparent client-side encryption works and present an abstract concept for the implementation process. After that, we present general limitations of transparent client-side encryption.

Checking the authenticity of public keys

From Alice's point of view, there are two major problems to be solved in a client-side encrypted cloud storage system:

1. Making decryption of the cloud storage possible on each of Alice's devices without extracting the private key from the first device.

2. When granting Bob access to files, Alice needs to get Bob's public key authentically without trusting the provider.

These problems can also be seen as a single one. We can create a new key pair on every device we add. We regard Alice's additional devices like other cloud storage users. Thus, we only have to face the second problem. When Bob was granted access for one device, he still has to grant access to his other devices. At the end of this chapter, we describe how he can do this.

This problem also had to be faced by established cloud storage providers. Duane C. Wilson and Giuseppe Ateniese [3] examined three such providers (Wuala, Tresorit and Spider Oak). They found out that all three providers worked as Certificate Authorities (CAs), which were used to generate trust in the public keys of the users. This is just the same approach we saw when examining Boxcryptor (see chapter 2.2). With this approach, man-in-the-middle attacks are easy to perform for the cloud storage provider, as described in chapter 2.1.

There are two famous ways of authentically providing a public key:

1. Hierarchical Public-Key Infrastructure: Trust is created by Certificate Authorities (CAs), which check the ownership of the private/public key following policies. This approach is expensive, time-consuming and not wide-spread (concerning private end users).

2. Web of Trust: Trust is created by users verifying the ownership of other users' public/private keys. Trust can also be transitive, if A trusts B and B trusts C, then A might also trust C. The web of trust approach is, for example, used in GNU Privacy Guard (GnuPG)[13].

[13] `https://www.gnupg.org/`

We assume, that users do not have much knowledge about cryptography and do not want to invest much effort. That is why the second approach is the better one for our use case.

Verifying public keys can be simplified by the use of smartphones. An example is the messenger Signal[14], which allows users to verify public keys by using a QR scanner. Another example of using QR codes to verify public keys is the procedure described in the GNUnet tutorial[15]. Also OpenKeyChain[16] uses this procedure.

As the maximum capacity of a QR code is 2953 Bytes[17], not every public key can be encoded (for example, my 4096 Bit RSA public key[18] has a size of 7199 Bytes). That is why the public key is not transferred directly via the QR code. The public key is transferred using web of trust key servers. Of course, any other channel (e-mail, cloud, etc.) could be used for the transfer, as well.

After the download, the authenticity is verified by checking the key's fingerprint via QR code. The choice of the hash function used to create the fingerprint depends on the version of the key. For version 3 keys, it is MD5, for version 4 keys, it is SHA1 [22, pages 70 - 71]. As there has already been found a freestart collision for SHA1 [23], it would be better to switch to for example SHA256 in the future (according to [24, page 38]).

Of course, it is also possible to first scan the QR code and download the key afterwards. This way, getting and verifying the public key can be done in one step for the user. The first procedure is represented in figure 18, the second one in 19.

To give an example, it is possible to scan the QR code of my public key fingerprint from figure 20 using for example OpenKeyChain. Of course, the QR code in this document does not mean that you actually got my public key's fingerprint authentically. It depends on how you got this document. The QR code is authentic only if you got this document in an authentic way.

We combine the idea of one private key per device with the web of trust idea. By also adding the QR code public key verification, creating trustworthy public keys is possible in an easy way.

When Alice adds her first device to her client-side encrypted cloud storage, her client software creates a key pair. A good choice for the first device is a mobile device, like a smartphone. A mobile device has a camera, and a QR code scanner can be used to easily verify public keys. By using a mobile device as the first device, no keys have to be compared byte by byte when adding more devices.

On the other hand, a smartphone can easily be lost or stolen. That is why we should think about key revocation at this point. When creating a private key, we can also create a key revocation certificate. This should be stored in a safe place, and of course not on the smartphone, because then we cannot use it when we do not have

[14] `https://whispersystems.org/`

[15] `https://gnunet.org/first-steps-using-gnu-name-system`, paragraphs "Creating a Business Card" and "Be Social"

[16] `https://www.openkeychain.org`

[17] `https://archive.is/20120915040049/http://www.qrcode.com/en/aboutqr.html#selection-497.11-497.15`

[18] `https://pgp.mit.edu/pks/lookup?op=get&search=0x0F72F672A6506F46`

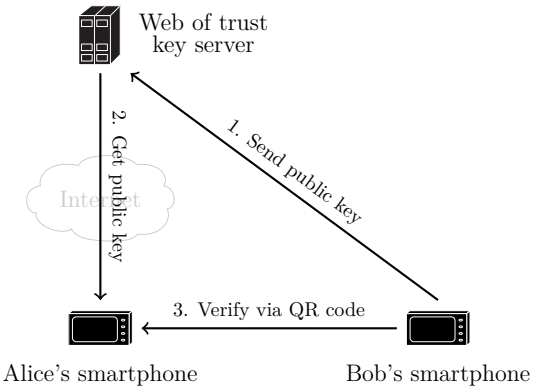

Figure 18: Alice downloads Bob's public key and verifies it afterwards.

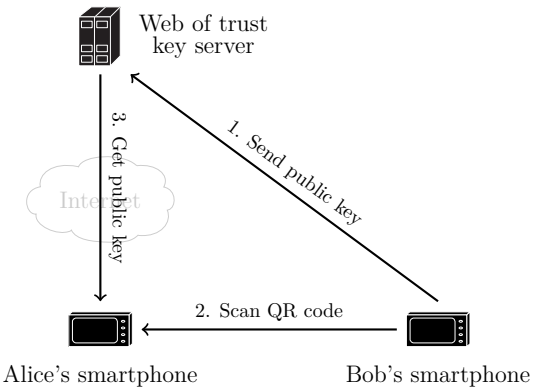

Figure 19: Alice scans Bob's public key's QR code, and performs downloading and verification in one step (from the user perspective). On her smartphone, of course, the verification is done after the download.

the smartphone anymore. A good idea is to burn it on a CD. As smartphones do not have a CD burner, we cannot do this directly. But we will establish a confidential and authentic channel to our notebook/desktop PC anyhow later in this procedure. Once we have it, we can transfer the revocation certificate and burn it on CD using the CD burner of the notebook/desktop PC.

It would also be possible to use a master key and subkeys. The master key could be stored on a smartcard. Then, the smartphone's key can still be revoked after the loosing the smartphone. To do so, we use the master key stored on the smartcard.

We denote the public key on Alice's first device K_1^p. Alice encrypts her files using the "File encryption/decryption system". The Clearance Key (CK) of the cloud storage folder is encrypted using K_1^p, we denote this $E_{K_1^p}(CK)$. A sketch of the procedure is shown in figure 21.

Figure 20: The QR code encoding my public key's fingerprint. The actual encoded data is *openpgp4fpr:4D891F4E173674D5CCD2BF9D0F72F672A6506F46*.

When Alice adds a second device, another key pair is created by the client software on it. We denote the public key of this key pair K_2^p. Alice transfers K_2^p authentically to her first device by using QR codes. The client software sends K_2^p to the web of trust key server. Alice downloads K_2^p to her first device. She then checks the authenticity of K_2^p using QR codes. This procedure is shown in figure 22. In the key verification software, the second device can be marked in red in case the key has not yet been verified. When the key has been verified, the device can be marked in green. It would also be possible to mark the device in yellow, when the same key/fingerprint was received via multiple untrusted channels (e.g. web of trust key servers, e-mail and SMS). Of course, this does not provide real authenticity. But on the other hand, an attacker has to put more effort into controlling multiple channels than into controlling only one channel. That is why it is still better than not verifying the key at all.

Alice can now be sure that K_2^p has authentically been transmitted to her first device. She then encrypts CK with K_2^p, denoted $E_{K_2^p}(CK)$, and uploads this ciphertext to the cloud storage. The second device authenticates to the cloud storage provider and downloads the ciphertext as shown in figure 23. By decrypting the ciphertext with the private key, the second device also sees the plaintext of the files.

For this scenario, a one-way authentic channel is sufficient. Anyway, Alice might lose her smartphone and buy a new one. If so, we have to check the new smartphone's public key from the second device (which might not have a camera). How we can do that using only QR codes is described in the segment "Establishing secure channels with only one QR code authenticity check".

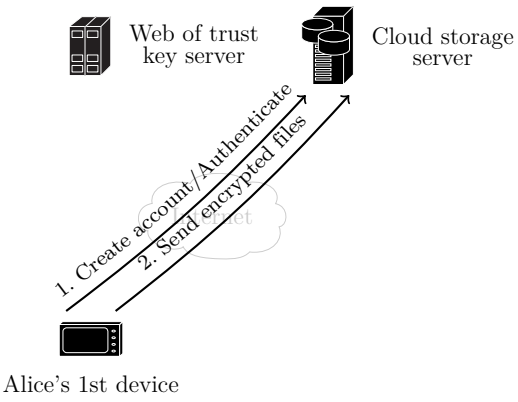

Figure 21: Alice adds her first device to her client-side encrypted cloud storage.

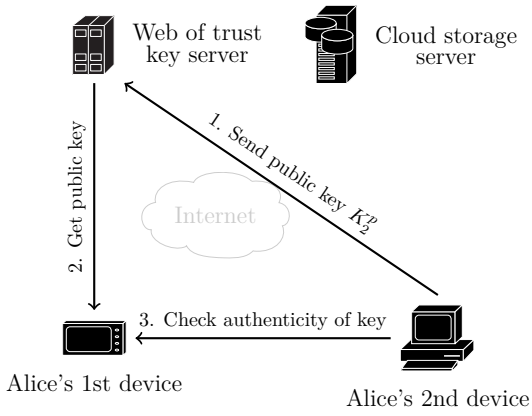

Figure 22: Alice adds a second device and checks the authenticity of the public key.

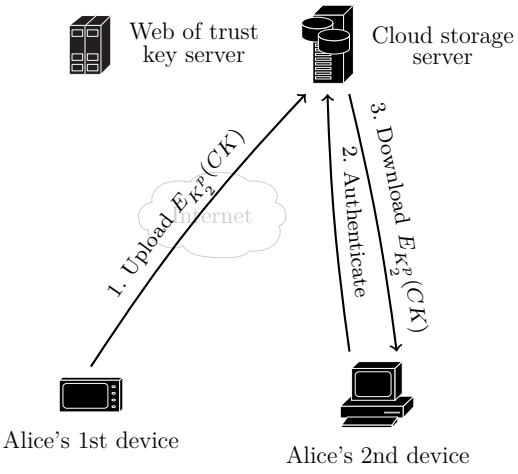

Figure 23: Alice's 2nd device receives all data needed for the decryption.

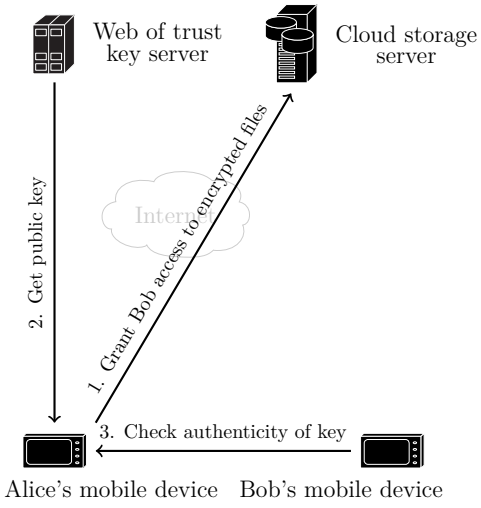

Figure 24: First steps of Alice sharing encrypted files with Bob.

When Alice wants to share a folder with Bob, she first grants Bob access to the folder on the cloud storage server. Then Alice and Bob meet and use their smartphones to transfer Bob's public key to Alice's smartphone in an authentic way (QR codes), as shown in figure 24. In the key verification software, we once more use the red and green coloring mentioned when Alice added her second device.

Afterwards, Alice proceeds just like adding a new own device, i.e. by uploading the encrypted clearance key of the shared folder. Bob's mobile device could then get the encrypted clearance key and decrypt it.

To also add Bob's other devices, the software on Bob's mobile device encrypts the clearance key with the public keys of the other devices. It then uploads the encrypted keys to the cloud storage, where they are downloaded by Bob's other devices. This can be done even if Bob has read-only access. The encrypted keys would then be stored in another place inside Bob's cloud storage folder, where Bob has write access.

For this approach, it is required that Bob got the public keys of his other devices in an authentic way on his mobile device. He can do that by displaying a QR code on his PC/notebook/tablet and scanning it with his mobile device. This way, he never has to read a public key fingerprint byte by byte. The verification of the public keys is done by the software on his mobile device. It is comfortable and not prone to human reading errors.

Establishing secure channels with only one QR code authenticity check

As already mentioned, we need to verify the transferred public keys when using asymmetric cryptography to establish a secure channel between two devices. We assume that the public keys have already been transferred, e.g. via a web of trust key server. If both devices have a monitor and a camera, e.g. are smartphones, verifying is easy. We display the QR code of the public key's fingerprint on one device and read it via the camera of the second device. We then proceed vice versa.

If none of the devices has a camera, we cannot use this approach. We can then compare the fingerprints byte by byte. Another approach is to create colored pictures of the public keys, which can easier be compared by humans[19].

If one of the devices has a camera, we can still establish the secure channel in an easy way. We could, of course, verify one public key via QR code and the other one by comparing the fingerprint byte by byte, but that would not be comfortable.

Let us assume that we want to establish a secure channel between a smartphone and a desktop PC. We display the QR code of the public key's fingerprint on the desktop PC's monitor and verify it by using the smartphone's camera.

Next, we verify the smartphone's public key on the desktop PC. We could do that by comparing the fingerprint byte by byte. But that is not a comfortable way.

On the desktop PC's monitor, we display a QR code containing the smartphone's public key fingerprint and a random number (e.g. 4 digits). We then scan the QR code with the smartphone's camera and verify the smartphone's public key on the

[19] `https://telegram.org/faq#q-whats-this-encryption-key-39-thing`, example picture available on `https://telegram.org/img/key_image.jpg`

smartphone. If it is correct, we are done and could terminate the procedure. But then we would not make sure that the user really checked the smartphone's public key. To ensure that the public key was verified on the smartphone, we ask for the random number on the PC. When the public key verification on the smartphone was successful, the random number is displayed. If the user does not enter the correct number, the procedure is aborted. This way, we can be sure that the user really did verify the public key on the smartphone.

The length of the number is not relevant for cryptographic purposes. The number makes it harder for the user to misbehave. The procedure is represented in figure 25.

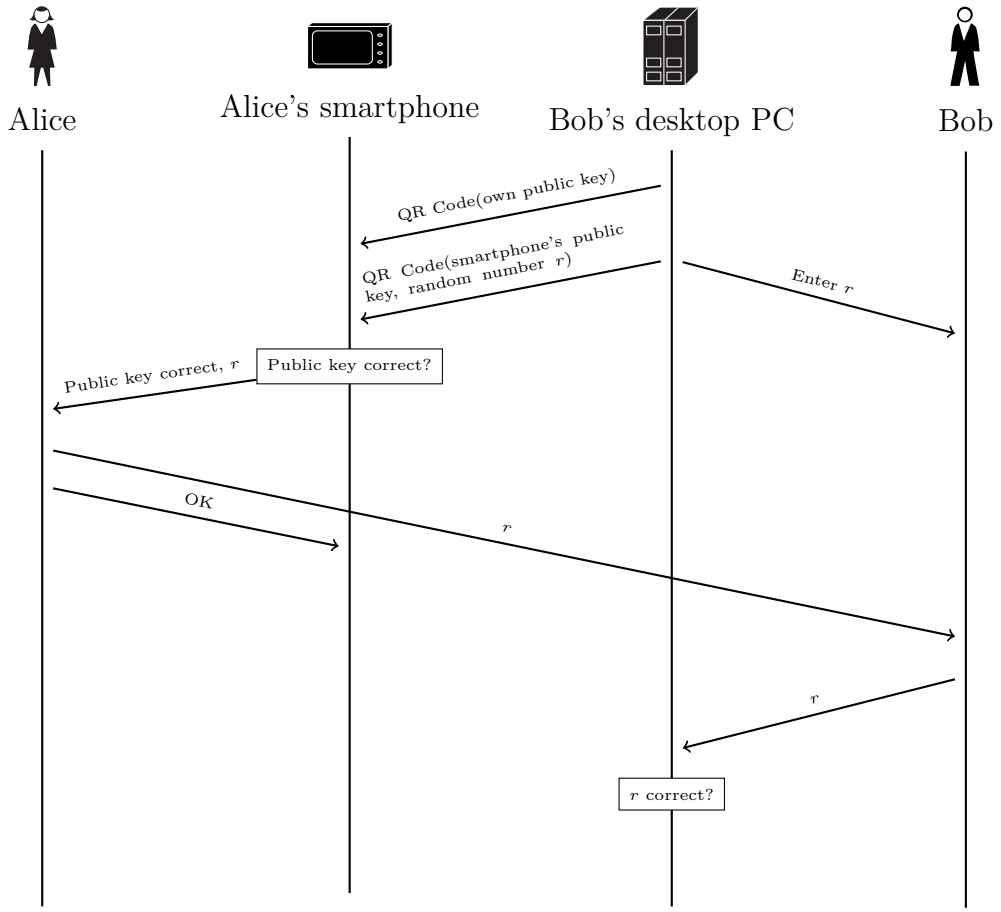

Figure 25: Alice and Bob establish a secure channel with only one QR code verification. Alice uses her smartphone, Bob uses his desktop PC.

If we try to perform a man-in-the-middle attack on the protocol, we need to substitute at least one of the two public keys. Let us assume that we substituted the desktop PC's public key. Then the smartphone will not verify it when it scans the corresponding QR code from the desktop PC's monitor. So let us assume that we substituted the smartphone's public key. But then, the smartphone will not verify it when it scans the corresponding QR code from the desktop PC's monitor. The procedure will be

aborted on the smartphone. It will also be aborted on the desktop PC, as the user does not type in the random number. Hence, we cannot perform a man-in-the-middle attack on this protocol as described here.

For the verification of the smartphone's public key, the users only need to transfer the (short, e.g. 4 digits) random number from the smartphone to the desktop PC. This is easier than comparing the smartphone's public key fingerprint byte by byte. Besides, Alice does not need to know that she must compare fingerprints. She can just follow the interactive protocol which tells her what to do next.

In the last two segments, we have seen how public key verification can be done comfortably by the users. We described how users can conduct every needed public key verification using QR codes and a camera. We also presented a protocol, that works with QR codes, even when one of the devices does not have a camera. If users follow these approaches, the man-in-the-middle attack described in chapter 2.1 cannot be performed anymore.

Securing cloud storages (non-)transparently

In the following segments, we treat the transparency part. That means, we show how we can integrate our security measures into the existing cloud storage system without the user noticing.

In this segment, we present non-transparent and transparent encryption. We discuss the pros and cons and name the requirements for transparent encryption. We decide to elaborate a system with transparent encryption, as it is more comfortable for the user and its requirements are satisfied on many systems.

When securing a cloud storage service, we can use an additional folder or do it in a transparent way. Transparently means that the user still uses the same folder as before and is (in the first place) not aware of any changes.

The approach of adding a folder is shown in figure 26. An advantage of this solution is, that it is compatible with every operating system. There are no special requirements the operating system has to satisfy. A disadvantage is, that we have to hide the cloud storage directory from our user Alice, because she could accidentally move her files into that folder. That can also mean that we have to hide the cloud storage program from her, because it could open the cloud storage directory. This can mean a lot of effort. We could also use the cloud storage provider's API. Anyhow, this adds a requirement. The provider would have to provide that API. Additionally, our solution would then be specific for the provider.

The second possibility (the transparent way) is shown in figure 27. An advantage of this solution is that the user can directly use the cloud storage directory. We do not have to hide it or the cloud storage program. A disadvantage is that the operating system has to meet a requirement. It must allow us to gain control over the cloud storage directory.

We take the second approach, i.e. we encrypt the cloud storage in a transparent way. We take this decision because of the stated advantage. The disadvantage is not a decisive factor, because this requirement is often satisfied. Linux allows us to gain

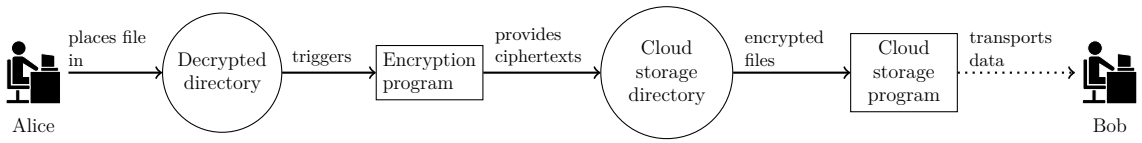

Figure 26: Non-transparent encryption of a cloud storage. The dotted line represents a communication via various routers in the internet. The decryption process on Bob's side is not shown.

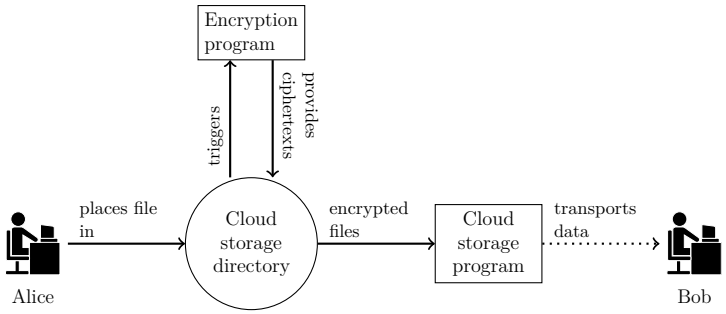

Figure 27: Transparent encryption of a cloud storage. The dotted line represents a communication via various routers in the internet. The decryption process on Bob's side is not shown.

control over the cloud storage directory via Filesystem in Userspace (FUSE)[20]. On Mac OS X, there has been the MacFUSE[21] project which is no longer maintained. On Windows, Dokany[22] is available. As creating file systems in userspace is a feature desired by application developers, it is likely to be developed for other operating systems, too.

We presented transparent and non-transparent encryption of cloud storages. As we have seen, transparent encryption provides better usability than non-transparent encryption. As a requirement, the operating system needs to let us gain control over the cloud storage folder. This requirement is satisfied on many systems. That is why we decided to elaborate a transparently client-side encrypting system.

Securing existing communication procedures

In this segment, we analyze if it is possible to elaborate a transparently client-side encrypting system. We will conclude that it is possible indeed. We compare cloud storages to other types of digital communication (namely stunnel and e-mail). By doing this, we show general approaches to the problem of securing existing communication

[20] `https://github.com/libfuse/libfuse`
[21] `https://code.google.com/archive/p/macfuse/`
[22] `https://github.com/dokan-dev/dokany`

procedures. We also show general limitations and what requirements have to be fulfilled to secure a communication transparently.

We distinguish direct communication from indirect communication. Indirect communication means that the data is stored on a server and can be accessed at a later date. Direct communication means that the data is not stored on any server. The two parties are communicating directly with each other, receiving happens just after sending.

Stunnel is a multi-platform program, that encrypts TCP/IP communication. The two communication parties have a direct connection to each other. Stunnel provides the SSL/TLS support if one or both parties do not natively support it, see figure 28. Stunnel works as a proxy. That means, the sending program just sends its messages as usual on a specific port. Stunnel captures the packages on the port, encrypts and authenticates them and then relays the messages. This is possible, because stunnel does an SSL/TLS handshake with the receiving side (and the other side expects an SSL/TLS handshake). The advantage of this approach is transparency. For the sending program and the user everything works as usual. Also, the original program was not changed or extended.

Stunnel is an example, in which transparency works perfectly. The reason for that is the multitier architecture of the TCP/IP communication. The program, which creates the messages, uses the TCP/IP implementation to send them. The program on the other side of the communication channel uses the TCP/IP implementation to receive the messages. Both sides do not notice if the transfer happens in an encrypted way. The transparency is possible, because we can intervene on every involved device before the messages are sent or received.

In the e-mail case, no truly transparent solution exists to the best of our knowledge. When sending an e-mail, we communicate indirectly. Alice does not send her e-mail directly to Bob, but via at least one mail server.

We could try to encrypt e-mails like the stunnel communication via a proxy. But e-mail programs usually encrypt the connection to the e-mail server (maybe even via stunnel). For security reasons, it is a good idea to do so. If we encrypt the communication again, the server will not be able to decrypt it and cannot process it.

The encryption has to be done the other way around. Before we send the e-mail, it has to be encrypted. So if we regard the e-mail program as a black box, it does only start to communicate when it is already too late for us to intervene. The problem is that the creation and the transport of the mail is done by the same program.

That is why we have to change/extend the e-mail program to provide an end-to-end-encryption, see figure 29. This has been done for the famous open source e-mail program Thunderbird[23]. It has been extended by the add-on Enigmail[24].

E-mail communication is a case in which transparent encryption does not work. The reason for this lies in the fact that we can only intervene in the communication after the sending device has already established an encrypted connection to another device. There is no point at which we can get the e-mail in plaintext and encrypt it for further

[23] `https://www.mozilla.org/en-US/thunderbird/`

[24] `https://addons.mozilla.org/en-US/thunderbird/addon/enigmail/`

usage. That is why we have to change the original program via an add-on.

In the cloud storage case, we have an indirect communication like in the e-mail case. Anyhow, creation and transport of the messages are not being done by the same program. The creation of the message (file) can be done by any program, the transport is done by the cloud storage program. The interface between these two parts is a folder. All files that are created in some specific folder are uploaded to the cloud. This is our point of intervention. We gain control over the folder. We encrypt all files before they are moved to the specific folder, see figure 30. On the other side of the communication, we also gain control over the cloud storage folder. When the user accesses the folder, we decrypt the files and present the decrypted data.

Cloud storages can transparently be secured, because we can intervene after creation and before sending/receiving of the messages. We can do so by gaining control over the cloud storage folder.

As we have shown, not every existing communication procedure can be secured transparently. In the case of e-mail, it was not possible. In this case, security can be established by providing an add-on. In the cloud storage case, securing the communication transparently is possible. In the following segments, we abstractly describe how this can be implemented.

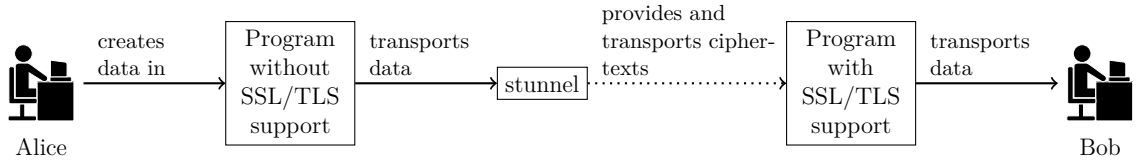

Figure 28: Outline of a communication secured via stunnel. If Bob's program also does not support SSL/TLS, Bob can use stunnel as well. The dotted line represents a communication via various routers in the internet.

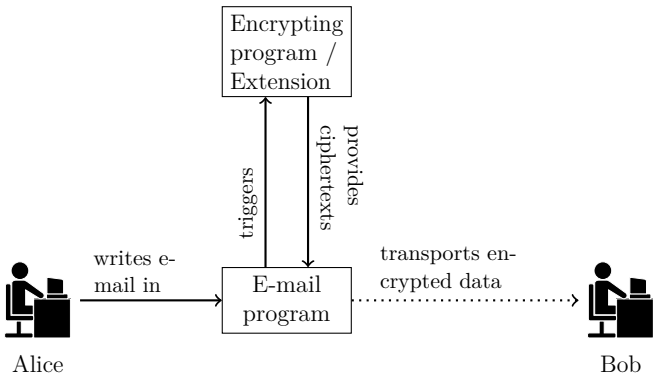

Figure 29: Outline of an encrypted e-mail communication. The dotted line represents a communication via various routers in the internet. The decryption process on Bob's side is not shown.

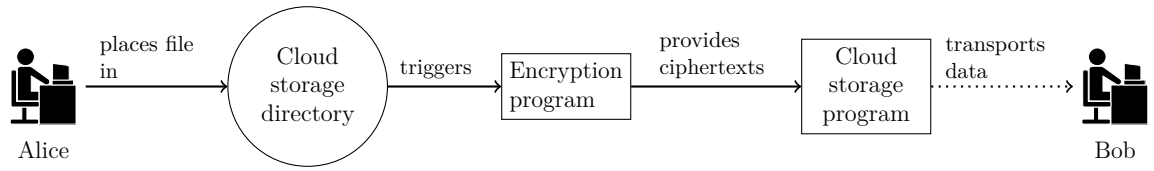

Figure 30: Outline of an encrypted cloud storage communication. The dotted line represents a communication via various routers in the internet. The decryption process on Bob's side is not shown.

Concept for the transparent way

In the last segment, we have seen that securing cloud storages transparently is possible. To implement it, we gain control over the cloud storage folder. In this segment, we show from an abstract point of view how we can do the transparent client-side encryption. Our concept is based on different views for different viewers. When the cloud storage program accesses the cloud storage folder, it is supposed to see the encrypted files. When Alice accesses the cloud storage folder, she is supposed to see the decrypted files.

We now describe the setup on Alice's PC. The cloud storage program is installed and used in the usual way. The cloud storage directory is also used as without encryption. We just add some software and two folders to the existing system. We call the added folders the "encrypted folder" and the "decrypted folder". These two folders can be placed anywhere. They should be hidden from the user, as they are not intended for direct use. Our user Alice does not need to know about them, as she still uses the cloud storage directory. Hiding them is easy, as Alice is not aware of them and the cloud storage program does not reference them. The encrypted folder contains the encrypted files and folders, while the decrypted folder contains the decrypted files and folders.

As already described, we gain control over the cloud storage directory. Our goal is

to present a different view of that folder depending on the viewer. If the cloud storage program accesses the cloud storage directory, we redirect the access to the encrypted folder. The program then sees the encrypted files and folders as shown in figures 31 and 32. Even the file/folder names may be encrypted. The following accesses will then concern encrypted file/folder names, which will work transparently. The write accesses are done without encryption. This is because the cloud storage program might write its configuration to the folder, which is not synced.

If our user Alice accesses the cloud storage directory, we redirect her access to the decrypted folder. Alice then sees the decrypted files and folders as shown in figures 33 and 34. So Alice's following accesses will concern decrypted files (and possibly decrypted filenames). By these different views, only the encrypted files are uploaded to the cloud storage server, while the folder behaves just as without encryption from Alice's point of view.

To be able to differentiate between the cloud storage program and the user, we can use the operating system's user concept. We create a user "cloud storage program", which is used to run the cloud storage program. When an access to the cloud storage directory is done, we check which user demands access. If it is cloud storage program, we show the encrypted folder, else we show the decrypted folder. That means that the confidentiality of the files depends on the user concept of the operating system. If the cloud storage program is able to switch to another user, it can read the decrypted files. Anyhow, if our operating system has weaknesses, there might as well be other possibilities to get inside and read the files. In that case, we can still limit the damage by the approach presented in the second part, see section 3.

Of course, we need to confine the cloud storage program, as it is potentially proprietary software. For example, we need to prevent access to the decrypted folder. We describe measures to confine the program in chapter 2.4.

In this segment, we explained from an abstract point of view how transparent client-side encryption can be implemented. We used an encrypted and a decrypted folder to store the encrypted respectively decrypted files. When an access is made to the cloud storage folder, we redirect it to the encrypted or decrypted folder depending on the user who made the access. This way, the cloud storage program only sees encrypted files, while the user still sees the files as plaintext.

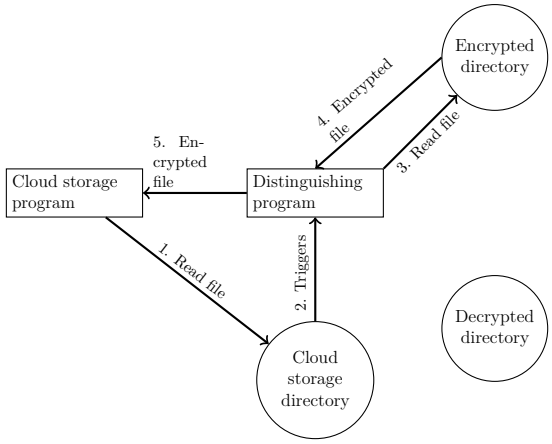

Figure 31: Transparent cloud storage program read access.

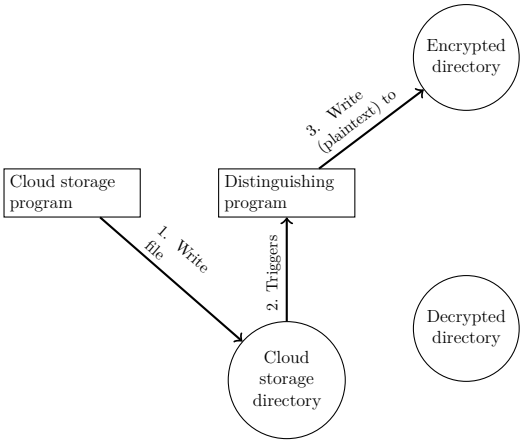

Figure 32: Transparent cloud storage program write access.

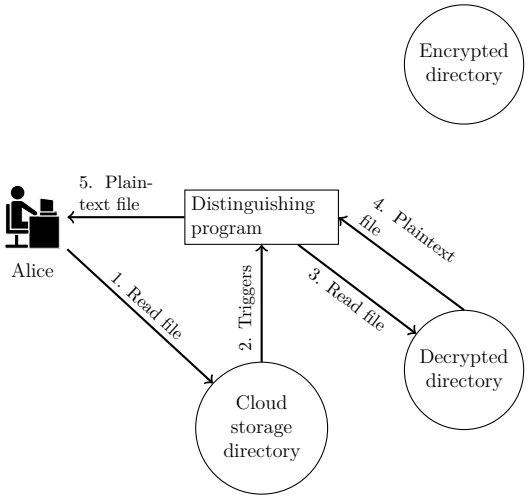

Figure 33: Transparent user read access.

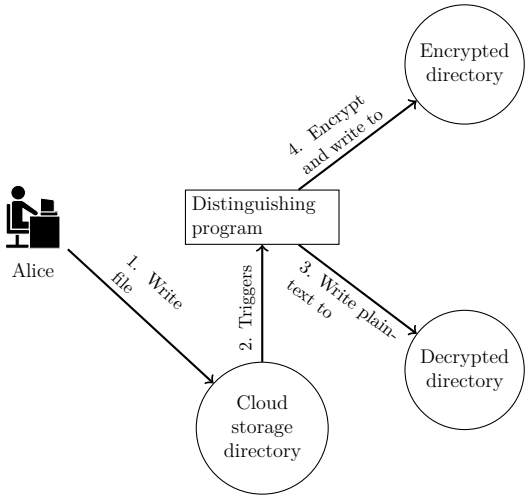

Figure 34: Transparent user write access.

Transparency limitations concerning shared folders

In the last segment, we presented our concept for transparent client-side encryption. This concept works perfectly for the daily usage of the cloud storage. Daily usage means dropping files in the cloud storage folder and accessing them from multiple devices. However, when we use functionality beyond that use case, there are limitations in transparency.

In this segment, we show that there is a general limitation to transparency when sharing folders and receiving shared folders. The reason for this is that we did not define an interface to do so. Defining the interface, however, would add a requirement to the behavior of the cloud storage application. We instead show how we can combine security and usability within the described limitations.

When receiving a shared folder, we should check if this folder really comes from the user we expect it to come from. Otherwise, the cloud storage provider would be able to plant faked documents or viruses on us. We can use signatures to do the check.

When checking if a message is authentic, there are two entities involved. One is the claimed sender, the other one is the signer. Only if claimed sender and signer match, the signature should be accepted. We explain the signature verification process by taking e-mail as an example.

An e-mail has a sender, which is contained in the e-mail header. This sender is easily interchangeable for the e-mail client and every server delivering the e-mail, as it is not bound to any kind of signature. We called this entity the claimed sender. If the e-mail is signed, it also contains a signature. The entity bound to the public key (by a certificate) is the signer. We give an example in figure 35.

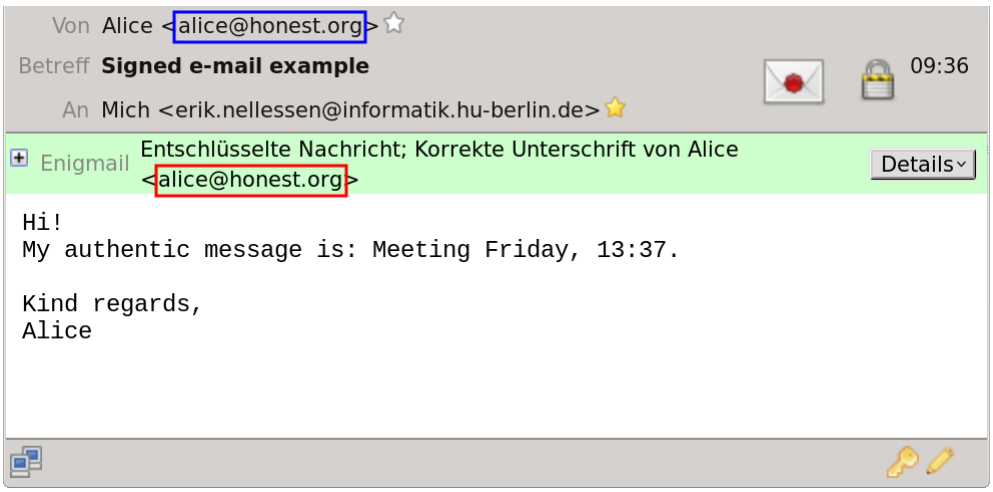

Figure 35: Example for a signed e-mail. The claimed sender is marked in blue, the signer is marked in red.

An e-mail client can check if claimed sender and signer match. If it does not do that, it is easily possible to fake authentic messages. We leave the claimed sender as it is and

sign our faked e-mail with any private key to which the receiver has a trusted certificate. The receiver's e-mail client then checks the signature and accepts it. It does that because the signature is correct. The signer just does not match the claimed sender. Only if Alice compares claimed sender and signer with her own eyes, she can notice the forgery. Preventing this kind of attack is possible in e-mail programs, because both claimed sender and signer are available. As we can see in figure 36, Enigmail (1.9.2, Thunderbird 38.7.0, GnuPG 2.0.26) does not do that. The described attack is possible[25]. We additionally tested evolution[26] 3.12.11, the attack was also possible with this software.

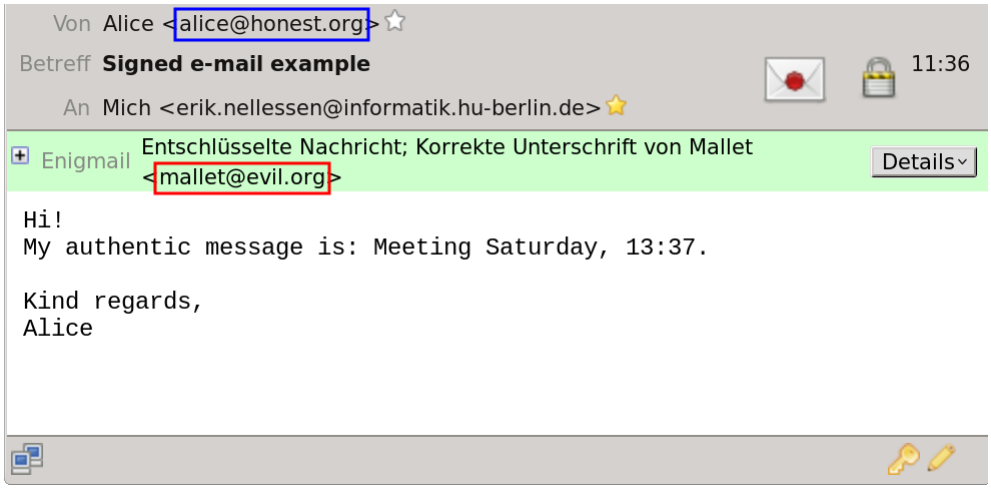

Figure 36: Example for a signed e-mail with different signer and claimed sender. The claimed sender is marked in blue, the signer is marked in red. As we can see, Mallet changed the content of the message. It claims, that the meeting is on saturday instead of friday. Only if we compare claimed sender and signer, we see that something is wrong. Enigmail could do this automatically and warn us, but this is obviously not implemented.

In the cloud storage use case, we also would like to check if claimed sender and signer match, because we also want to prevent the described attack. Getting the signer is easy. When sharing a folder, we place a signed file in the folder which contains the folder name. We also place a signed and encrypted file in the folder containing the password for decrypting the folder. When accessing the shared folder, we check the signatures and know the signer.

In the cloud storage use case, we did not mention the claimed sender yet. When sharing a folder, we have to do *two* steps. Step one is to enable decryption for the other person (by placing the encrypted password file in the folder). Step two is to enable access for the other person at the cloud storage provider. We already described this

[25]Request for enhancement: `https://sourceforge.net/p/enigmail/bugs/612/`
[26]`https://wiki.gnome.org/Apps/Evolution`

procedure in the segment "Checking the authenticity of public keys", see also figures 22, 23 and 24. When accessing a shared folder, the steps are done in reversed order. We first accept the shared folder. In that step, the claimed sender comes into play. The cloud storage provider presents a sender, of course not authentically. In the second step, we verify the signature and obtain the signer. If we want to check signatures like in the e-mail use case, we need to check if the claimed sender and the signer match.

Sharing the folders in two steps already breaks transparency. Alice knows about our program, because she now has to do two steps instead of one when sharing a folder. We could of course try to intervene in the cloud storage provider's user interface and do both steps in one user operation. But still then, Alice would have to use our program instead of the original cloud storage program. We could hack the original program to do what we want. But this would mean a lot of effort and would make our solution very specific.

It is also very hard to check claimed sender and signer transparently. It is not clear in what way the cloud storage software allows us to share folders/accept shared folders. For example, Dropbox does this via a website[27]. But there are a lot of other ways possible (e.g. e-mail). We can, of course, try to intervene in the cloud storage provider's user interface and get the claimed sender from there. But this would add a requirement to the cloud storage provider, namely to have a special kind of user interface. Additionally, our solution would then be specific for a provider.

A possible way to still prevent the described attack is to let the user accept the shared folder in whatever way the cloud storage application does it. Afterwards, we present the signer (via a Graphical User Interface (GUI)) and ask our user Alice, if she accepts the folder from that signer.

In this segment, we showed that it is not a good idea to keep transparency in the file sharing use case. Implementing a truly transparent approach would be very complex and would make our solution specific for one provider. Adding more providers would again mean great effort. That is why we decided to let our user do two steps when sharing a folder and check the signer when receiving a folder. This way, our software is not truly transparent in this use case, but is a general, not provider specific solution.

Transparency limitations depending on the "File encryption/decryption system"

In this segment, we show additional limitations concerning transparency. We examine, what effects the design of the "File encryption/decryption system" has on the transparency of the resulting client-side encryption. The main statement is that the cloud storage folder might be accessed by interfaces not manipulable by the "File encryption/decryption system". That is why the user might see the encrypted file structure. The use cases we examine are sharing files and restoring old file versions.

The "File encryption/decryption system" might do strange things like dividing files into blocks etc. This is not a problem as long as the "File encryption/decryption system" is the only way the cloud storage is accessed. This is the case for the simple

[27] `https://www.dropbox.com/help/19`

usage. Whenever Alice tries to access her files, we present the decrypted view via the "File encryption/decryption system".

Anyhow, Alice might access the cloud storage in other ways (like a web interface). This might happen to share files or to restore older file versions. Then, Alice sees the encrypted version of her files. Filenames might be encrypted or the files may be divided into multiple blocks, maybe not even a folder structure exists.

Just like in the segment "Transparency limitations concerning shared folders", we once again could try to change the interface. For the same reasons as in that segment (less effort, more general solution), we do not do that. Instead, we now describe measures to help the user deal with the encrypted folder structure. A program which shows decrypted folder structures in the browser instead of encrypted ones actually exists. It is called EncFSAnywhere[28]. It works for file structures encrypted with Encrypted Filesystem (EncFS). As we will see in chapter 2.4, this is the program we will use for encryption.

When Alice wants to share a folder, in the first step she has to enable decryption for the other user(s). This is done via a program provided by us. This program can access the files via the "File encryption/decryption system". If the cloud storage application provides interfaces, it can also share the folder at the cloud storage provider. In every case, our application can tell the user what files/folder have to be shared at the cloud storage provider. This can for example be required if the file names are encrypted. When accepting a shared folder, the user has to accept the encrypted folder name in the first step. In the second step, we can present both the encrypted and the decrypted folder name.

When Alice wants to restore an old file version, she also might have to use something like a web interface. She then would have to identify the old file name by looking at the encrypted file structure. We can help her by providing a program which gets a decrypted path as input and provides the encrypted path as output. It is also possible to implement a program which works in the browser and provides the decrypted file structure for the user. Anyhow, this program would have to make assumptions about the provider's web interface. So such a program is not a general solution.

In some way, the "File encryption/decryption system" has to match the cloud storage provider for use cases like sharing files and restoring old file versions. If, for example, the "File encryption/decryption system" saves all files in only one container, the cloud storage provider should support sharing only parts of the container. Else, the user would have to share the whole container when only wanting to share one file. The receiving user would then have to download the whole container. If the user wants to restore old file versions, restoring only parts of the container should be supported. So to enable use cases like sharing folders or restoring old file versions, a simple "File encryption/decryption system" working with files and folders like e.g. Cryptree, is a good choice.

We showed that the "File encryption/decryption system" influences the degree of transparency. Just like in the last segment, implementing a truly transparent approach

[28] `https://bitbucket.org/marcoschulte/encfsanywhere`

would be very complex and would make our solution specific for one provider. Instead we showed, how the user can be helped when sharing files or restoring old file versions. This way, we have less implementation effort and provide a more general solution.

Revoking access to a shared folder

In this segment, we face the problem of revoking access to a shared folder. That means that we shared a folder with Bob, but now do not want Bob to have access to that folder anymore. We integrate this problem into our work and conclude that it is not part of our focus. We provide references, which show in detail how the problem can be solved efficiently.

Revoking access to a shared folder also has to be done in two steps. Let us assume that we want to revoke the access for the user Bob. In the first step, we revoke the access at the cloud storage provider. That means that Bob cannot access the contents of the folder anymore, but only if the cloud storage provider really behaves like we want it to. Besides, Bob and the cloud storage provider could work together to get access to the plain files in the folder. The cloud storage provider would contribute the access to the encrypted files and Bob would contribute the decryption keys.

That is why we need to do the second step as well. In the second step, we revoke the access cryptographically. That means that we have to use another key to encrypt the files in the folder. Using new keys only for changes or new files is called lazy revocation. Reencrypting everything with a new key is a secure, but not very efficient solution.

This problem is discussed intensely in already cited papers concerning the "File encryption/decryption system", see chapter 2.2. As we do not focus on that part of the system, we do not discuss it any further.

Conclusions

In this chapter, we presented our system design. In the first two segments, we showed, how public key verification can be done securely and comfortably. We mentioned the required public key verifications in cloud storage use cases. We then described, how users can do all of them by scanning QR codes with their smartphones. We also provided a protocol, which uses QR code scanning even if one device does not have a camera.

In the latter segments, we discussed the pros and cons of transparent client-side encryption. We decided to encrypt transparently. We discussed if transparent encryption was possible and described abstractly how it can be implemented. Lastly, we discussed the general limitations of this approach.

2.4 Implementation

We implemented the system described in chapter 2.3 to allow transparent client-side encryption with user-done public key verification when using Dropbox. Instead of the cloud storage provider, users verify each other's public keys using QR codes and

smartphones. This way, the man-in-the-middle attack as described in chapter 2.1 is not possible anymore. At the same time, the public key verification can be done comfortably. The encryption of the cloud storage is done transparently. That means that users still use their cloud storage folder just like in the non-encrypted setting. The implementation can be found on github[25] (a tutorial on how to use the software can be found in the Readme in the appendix of this work).

The implementation is not specific for Dropbox. We can easily use other cloud storage applications. As shown in chapter 2.3, we solved the problem of transparent client-side encryption with user-done public key verification independent of the cloud storage application. That means that even open source cloud storage applications which do their best to preserve the user's privacy do not need to implement a client-side encryption (in the limitations described in chapter 2.3). An example for such an application is Cloudfusion[29]. Cloudfusion is an open source application allowing to access well-known cloud providers.

Checking the authenticity of public keys

We now analyze how the public key verifications as described in chapter 2.3 can be done in practice. We will conclude that the required software already exists and does not need to be implemented. It can be modified to support two-way public key verification with only one camera we described in chapter 2.3.

Public key verification is a general problem. It is not specific for the cloud storage encryption use case. That is why there are programs, which solve this problem in general. GnuPG[30] is such a program. It is an open source implementation of the Open Pretty Good Privacy (OpenPGP)[22] standard. The OpenPGP standard defines formats for encrypted data, signed data and certificates.

GnuPG has features, which are not included in OpenPGP:

- GnuPG supports main keys which are only used to sign certificates. They are not used to decrypt or sign data. They sign certificates for subkeys which are used for decryption and signing. As the main keys are not often used, they can be stored more secure (with more effort to use them) than the subkeys.

- GnuPG supports certificate validation via a web of trust. The authenticity of public keys is verified and signed by the members of the web of trust. Besides, transitive trust is possible. That means that if B signed the certificate of C, and A trusts B, A might also trust the certificate of C.

GnuPG does not support displaying QR codes of public key fingerprints. But we can easily create QR codes from fingerprints by using qrencode[31]. We can then use an image viewer to display the picture of the QR code. We gave an example in figure 20.

[29] `https://github.com/joe42/CloudFusion`, no progress since January 2015
[30] See footnote 13.
[31] `https://fukuchi.org/works/qrencode/`

The GnuPG equivalent program for Android is OpenKeyChain[32]. OpenKeyChain allows importing public keys via QR codes. To do so, it gets the fingerprint via the QR code. Then it searches the OpenPGP key servers for keys matching the fingerprint. The calculation of the fingerprint depends on the key version [22, pages 70 - 71]. For version 3 keys, MD5[26] was used, which is deprecated now. But even for version 4 keys, only SHA-1[27] is used. SHA-1 should not be used anymore according to [24, page 38]. Of course, it would also be possible to execute the procedure described here with a currently safe hash function like SHA-256[27].

From an abstract point of view, it would of course also be possible to transfer the whole public key by scanning multiple QR codes. This would imply more effort for the user. On the other side, the public key does not need to be uploaded. This way, the web of trust cannot harm the user's privacy (by showing connections to other users).

Using OpenKeyChain, we can also download the key first and verify it via QR code afterwards. Figure 37 shows the involved software components in a public key verification.

So the software for the public key verification as described in chapter 2.3 already exists and does not need to be implemented.

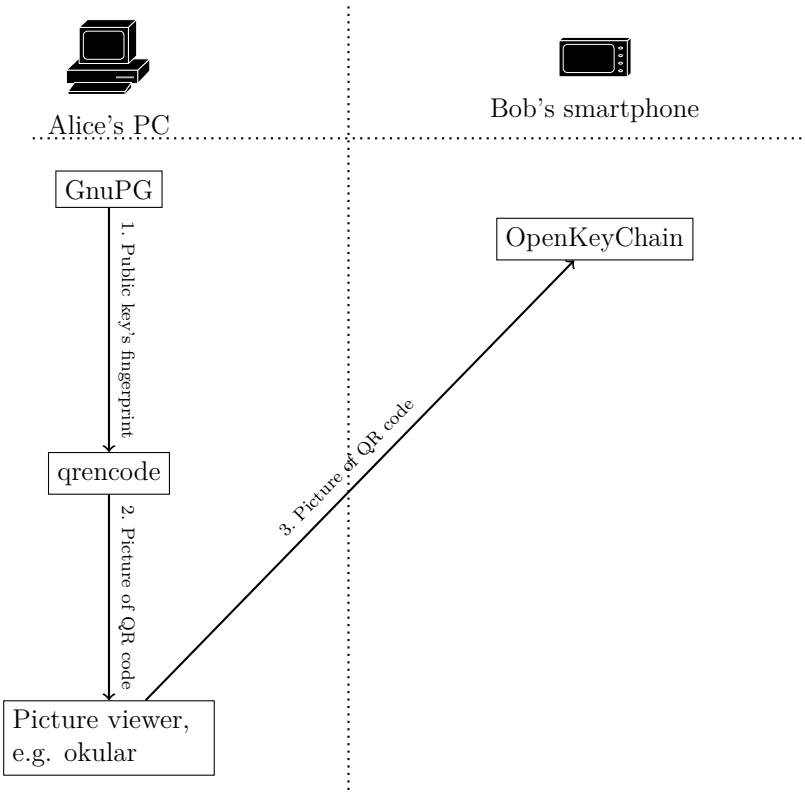

Figure 37: Involved software when verifying a public key.

[32]See footnote 16.

Transparent client-side encryption

We implemented the concept described in chapter 2.3. In this concept, two folders were added to the existing system. They were called the "encrypted folder" and the "decrypted folder". When accessing the cloud storage folder, Alice is redirected to the decrypted folder, while the cloud storage application is redirected to the encrypted folder. Our implementation was tested with the cloud storage provider Dropbox.

We used FUSE[33] to implement a virtual file system. This way, we gained control over the cloud storage folder. We started Dropbox as user "dropbox". When a process accesses the cloud storage folder, FUSE calls the functions defined in our program. We can then find out, which user requested the access[34]. Depending on the user, we redirect the access to the encrypted or decrypted directory.

When we start Dropbox as user "dropbox", we cannot access the normal user's display. We have to allow the access as the normal user. A "quick and dirty" (and insecure) way to do this is to use "xhost +"[35][36].

As stated in chapter 2.2, we used some of the ideas from Cryptree to implement a simple "File encryption/decryption system". An important element of our "File encryption/decryption system" is EncFS[37]. EncFS is a virtual file system which encrypts files stored in a conventional file system. It uses FUSE to present the decrypted files in a transparent way.

There has been a security audit[38] showing weaknesses in EncFS. Solving these security problems is not a concern of this work. We see EncFS only as one element to create a simple "File encryption/decryption system". And as already stated, even the "File encryption/decryption system" is not our main concern. Our main concern is providing secure and simple public key verification and transparent client-side encryption.

We now describe how our program works. We use EncFS to encrypt folders. We start with the top folder. First, we create a password using the makepasswd[39] program. We then start EncFS for the top folder using the password we just created. EncFS will encrypt all files, which we save to the decrypted folder and save the result in the encrypted folder. That means that we do not have to do step 4 from figure 34, EncFS does it automatically for us when we perform step 3.

We encrypt the password using our public key using GnuPG and save the result in the encrypted folder. When we terminate our program, we can decrypt the password file from the encrypted folder using our private key.

---

[33] See footnote 20.

[34] `https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201001/homework/fuse/fuse_doc.html`, paragraph "Getting FUSE Context"

[35] `http://linux.die.net/man/1/xhost`

[36] For a discussion on cleaner and safer solutions, see `http://serverfault.com/questions/51005/how-to-use-xauth-to-run-graphical-application-via-other-user-on-linux`

[37] `https://vgough.github.io/encfs/`

[38] `https://defuse.ca/audits/encfs.htm`

[39] `https://packages.debian.org/en/jessie/makepasswd`

To do so, we use GnuPG Made Easy (GPGME)[40]. GPGME is a library providing an API to perform encryption/decryption operations using GnuPG. We can then start EncFS with the password. It will decrypt the files we access in the decrypted folder.

We start EncFS as a process. We do so, because EncFS does not provide an interface to be used as a library. Starting EncFS as a process means that the Dropbox process might be able to gain information about the parameters the process was started with. That could be the password or an encrypted folder name/decrypted folder name mapping. That is why it is a good idea to limit the Dropbox user's access on process information, e.g. by using setfacl[41] on /proc .

When a folder is created within an existing folder, we proceed as we did with the top folder with one exception. We do not encrypt the password with our public key. Instead, we just store the password file in the decrypted folder. EncFS will encrypt it and store the result in the encrypted folder. That way we implement the idea from Cryptree with intuitively shareable folders. If we can decrypt a folder, we can also decrypt all of its subfolders.

When we want to share a folder with Bob, we just encrypt its password with Bob's public key and sign it with our private key. We created a program (called `share_a_folder`) which takes a folder and a public key fingerprint as input and performs the stated operations. Bob can then decrypt the folder and all its subfolders and verify the signature. Of course, we also need to grant Bob access via the cloud storage provider.

Lastly, we need to confine the cloud storage application, because it might be proprietary software. To prevent access to certain folders, we can use setfacl. We can also chroot[42] to set a new root directory for the Dropbox user. This way, we can easily confine the access to many folders. Of course, all other Linux sandboxing tools are available for us. We could also use additional sandboxing tools, e.g. MBOX[28].

In this chapter, we showed how we implemented the system we abstractly described. We tried using existing software whenever possible. An overview of the used software is given in figure 38. Our proof of concept implementation shows that the system we described works in practice.

[40] `https://www.gnupg.org/related_software/gpgme/index.html`
[41] `http://linux.die.net/man/1/setfacl`
[42] `http://linux.die.net/man/1/chroot`

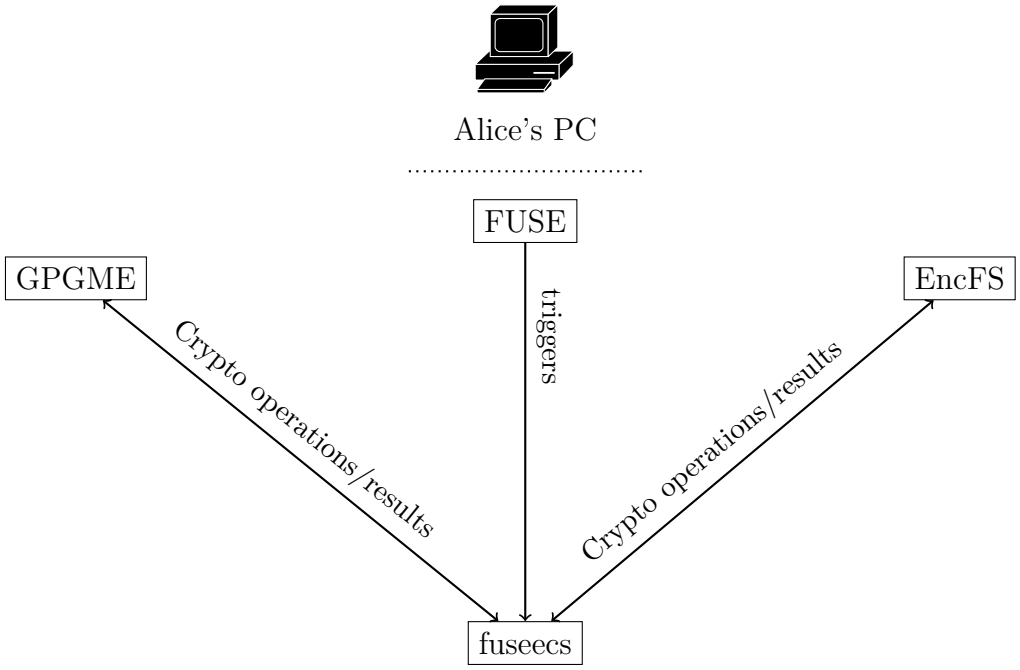

Figure 38: Involved software when transparently encrypting the cloud storage folder.

2.5 Remaining problems

After presenting our solution, we discuss remaining problems. These problems should be subject to further research. We explain how the cloud storage provider can still derive information from meta data. Anyhow, this problem exists in many types of encrypted digital communication (e.g. e-mail, instant messaging).

Even if we encrypt our files in the cloud storage, the cloud storage provider can derive a lot of information from meta data. This problem is not specific for the cloud storage use case. The same problem exists when sending encrypted e-mails or instant messages. The size of the message reveals, what kind of message (text, picture, video, audio) it is and how long it is approximately. But we do not have this problem when encrypting a local disk. All the encrypted data on the local disk appears as one single block if it cannot be decrypted (if we fill the disk with random data before using it, the block has the size of the disk capacity). As the cloud storage use case might be regarded as a special case of the local disk use case, there might be possibilities to make it harder to derive information from meta data.

The following information is visible to the cloud storage provider (depending on the "File encryption/decryption system"):

- Folder structures
- File sizes (up to the block size of the encryption algorithm, if it uses padding)

The provider can derive further information. We give some examples:

- File type: A lot of files of approximately the same size (about 2MB) in the same folder will probably be photos. A lot of files which vary more in size and have a size between 2 and 6MB will probably be MP3s. Files with a size of about 700MB, 4.7 GB or 25 GB will be disk images, probably of movies. A lot of small files could be source code.

- File collections: An attacker can create "fingerprints" of file collections to identify them in client-side encrypted cloud storages. File collections could for example be articles criticizing an oppressive government. To create the fingerprints, the attacker Mallet needs to have the file collection. He then encrypts it and saves the folder structure and the file sizes. Afterwards, he scans client-side encrypted cloud storages for users whose storages include file collections matching the fingerprint. He can then take measures to prosecute these users.

In other words, the privacy of users with a visible file structure is threatened. To protect privacy, the client-software could try to obscure the file structures by splitting files in blocks or merging files into big blocks.

Big blocks could be a problem when syncing the files. Depending on the encryption algorithm, much of the file could change when we change just a little amount of data. We would then have to upload way more data than would be needed. Many little files ("blocks") do not have that much uploading overhead. On the other hand, Mallet can see which blocks have been changed at what point in time and he can derive information from that. Big blocks do not prevent this attack if changes in files are logged, too. The attack is not implausible, as logging changes in files is often a feature in cloud storages. This legally collected data could be abused by an attacker. Deriving information from the behavior of the client could be complicated if the client would always write the blocks in an ascending order.

This could be done by using a virtual file system which behaves like a log-structured file system [29] or like a write-anywhere file layout [30]. We would save the encrypted files from the "File encryption/decryption system" in a virtual file system. Anyway, without additional encryption of that virtual file system, all of the information about file size and folder structure would again be visible to an attacker. If we encrypt the virtual file system, however, we cannot share files with other users without giving them the key to decrypt the virtual file system.

If we do it the other way around, i.e. first save the files in a virtual file system and then use the "File encryption/decryption system", the "File encryption/decryption system" will not work correctly because there are no folders anymore (assuming the "File encryption/decryption system" uses folders). A solution only seems possible when we combine encrypting files and obscuring the file structure in one system. Anyway, it is not clear whether the approaches presented here significantly increase the difficulty for an attacker to identify file structures.

Another idea is to not store all blocks on one provider, but to distribute them between several providers. This would make it more difficult for an attacker to attack the privacy of the user. However, it also increases the difficulty of sharing files with other users. That is because all users must then have a storage at each provider.

Anyway, connecting the encrypting program to multiple providers could be a good idea in the context of usability. We could do this in a way that is transparent to the user. Our user Alice would then have the experience of using only one program for all of her storages.

To this time, there is no system known to us which can prevent the described attacks. In all of the mentioned encrypted digital communication techniques (cloud storages, e-mail, instant messaging), attackers are able to derive information from meta data. This problem stays unsolved here and should be subject to further research.

2.6 Conclusions

In this part, we showed weaknesses in existing cloud storage systems and proposed a solution. We now present our conclusions, to show clearly, what can be learned from the work we have done.

First, we explained why it is necessary to create a cloud storage without trusted third parties. To do so, we showed in what way a cloud storage provider can perform a man-in-the-middle attack on cloud storage users. This attack results in the loss of the users' privacy.

We then analyzed existing cloud storage providers and existing research concerning client-side encrypted cloud storages. All examined cloud storage providers did not enable their users to use their system without the possibility of a man-in-the-middle attack. We also showed that the examined programs, that encrypt user data and use existing cloud storage providers, do not work transparently. The analysis of existing (theoretical) solutions for the "File encryption/decryption system" of the cloud storage system provided ideas for creating our own simple "File encryption/decryption system" (but we did not focus on that).

We described our solution in an abstract way. We addressed two main concerns:

1. Public key verification
2. Transparent client-side encryption

In the public key verification part, we described how smartphones simplify the process by scanning QR codes. This way, people do not have to compare public key fingerprints byte by byte. We also described a protocol to verify both public keys, when only one communication partner has a camera.

Transparent client-side encryption means that the user can use the cloud storage application in the same way as without encryption. In the transparent client-side encryption part, we examined what kind of applications can be secured afterwards without having to change the program. We concluded that cloud storage applications have this property. That means that cloud storage applications may be designed without client-side encryption (with the limitations mentioned). Even open source applications respecting privacy do not have to implement a client-side encryption. The client-side encryption can be implemented in another application without changing the cloud storage application.

We also showed that it is possible to do the client-side encryption in a transparent way, as long as the operating system lets us gain control over the cloud storage folder. Many systems satisfy this requirement by providing a functionality equal to FUSE. When sharing files, the user has to do one extra step (share the file in a cryptographic and in the cloud storage way). When accessing the files via another interface, e.g. a webinterface, the user sees an encrypted file structure. We referenced a program which claims to solve this problem.

We then presented our open source implementation of the described system. For the public key verification part, nearly all functionality needed is already implemented in OpenKeyChain. For the transparent client-side encryption part, we implemented our system on Linux using FUSE, EncFS, GnuPG and GPGME.

Lastly, we stated still existing problems which should be subject to further research. Attackers are able to derive information from meta data. Anyway, this problem also exists in the e-mail and instant messaging use case and is not specific to our system.

The contribution in this part of the work was to present a system preventing the man-in-the-middle attack described in chapter 2.1 while providing a high level of usability.

3 User-controlled decryption operations

3.1 Introduction

In the first part, we examined the status quo of cloud storage usage. We concluded that many users do not use client-side encryption. That means that their files are visible to their cloud storage provider. Privacy is lost for these users.

After showing weaknesses in existing client-side encryption solutions, we presented our system, which overcomes these weaknesses. Our system uses client-side encryption. That means that the users' files are encrypted before they are transferred to the cloud storage provider. That is why the cloud storage provider cannot see the file content. Client-side encryption is a requirement to use a cloud storage provider while preserving privacy.

When Alice uses our program and downloads the files from her cloud storage provider, the files are of course received in encrypted form. That is the only form in which the cloud storage provider has Alice's files. Anyway, we do not want to present Alice encrypted files, but files in plaintext. That is why our program has to decrypt her files before we present them to her.

For the decryption of Alice's files, a private key is needed. Only Alice is supposed to have that key. In the first part, we assumed that Alice's PC is trustworthy. That means that we expect her PC only to do things we want it to. We do not expect it to contain software or hardware which exposes the private key or plaintext files. That is why we could store the private key on Alice's hard disk drive, for example. Our program can then access the private key from Alice's hard disk drive and decrypt her files before presenting them to her.

We can also store the private key on a token. A token is a small device, e.g. a smartcard, that performs cryptographic operations. Tokens can increase usability and security of our system.

In this part, we add a token to the system described in section 2. We show, how a token should be integrated into the system to provide security and usability. Our assumption concerning the attacker Mallet is that he controls Alice's PC, but not her token. We conclude that letting the user confirm decryption operations on a trusted display increases the security gain achieved by adding the token. After that, we analyze what requirements a token should satisfy in our use case. We describe abstractly how displaying a file's meta data before decryption can be implemented. Our main idea is to bind a file's meta data to a decryption key by encrypting both in one blob. We also provide an open source implementation of the described system that uses a smartphone as a token.

Motivation

In the first part, we presented a system whose security only relies on the nondisclosure of the users' private key, assuming their hardware and software are trustworthy.

As the private key is critical for the security of the system, it can be protected against

misuse. Usually, a password is used to do that. When a password is used, accessing the hard disk drive or stealing the computer is not enough to misuse the private key. Anyhow, to provide this protection, the password must be strong. Strong passwords cut down usability.

Additionally, if Alice's hardware or software lose their trustworthiness, security is obviously not given for this system. If our attacker Mallet controls Alice's system, he simply gets her private key and decrypts the files or gets the decrypted files directly from Alice's system. Even a password cannot prevent this attack, as the password is entered on Alice's system and can be recorded by Mallet.

The first attack, i.e. Mallet stealing Alice's PC or accessing her hard disk drive, can be prevented by using a token. The private key is then generated and stored on the token. Alice's system can access the token only in a defined way. It can request operations to be done on the token. Figure 39 shows the general setup of this scenario.

Alice's PC/hard disk drive then no longer contain the private key. Mallet has to steal the token now. Anyhow, tokens can be protected against misuse by e.g. a Personal Identification Number (PIN) or biometric identification. As the token is protected against brute-force attacks, the PIN does not need to be as long and complicated as a secure password. This increases usability.

It is also a good idea not to have the private key in the PC's memory. Attacks reading the private key from the PC's memory (see e.g. [31]) can be prevented this way.

By accessing the token from each of her devices, Alice only needs one private key. She does not have to transfer her private key to each of the devices. She also does not have to do public key verifications for each device as described in chapter 2.3. This increases usability even more.

Anyway, only using a token just delays the attack in the second scenario (Mallet controlling Alice's PC). Mallet, who controls Alice's system, can just wait until Alice uses the token for the next time. This is likely to happen soon, as Alice needs the token to access her files. Once the token is present (and possibly unlocked by using a PIN or something similar), Mallet requests to decrypt the files he wants to see in plain. If Alice does not have an additional control about the operations done on the token, she does not notice this attack.

That is why only adding a token to the system does not provide a great gain in security. If Alice can control the operations done on her token, the security gain can be increased. Mallet getting the plaintext of a specific file f can then be delayed until Alice accesses f for the next time. That is why we describe and analyze ways of providing control over these operations for the user in this part of the work.

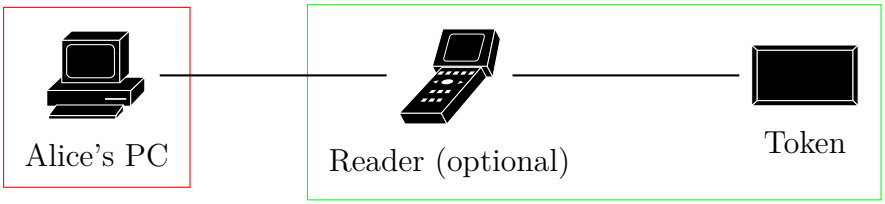

Figure 39: General setup. Devices with a high trust level are framed in green. Devices with a lower trust level are framed in red.

Problems addressed

The goals in our use case are usability and security. First, we define a set of requirements a token has to satisfy in our use case. We present existing types of tokens and analyze whether they meet our requirements. This way, we try to find a token which suits the needs in our use case.

Afterwards, we face the problem of secure meta data displaying. As already described, we show a file's f meta data to the user before the decryption is done on the token. We face the problem of making sure that the displayed meta data really leads to only decrypting f. We must make sure that we do not transfer decryption keys to the PC which can be used to decrypt other files. We somehow have to bind the decryption key to the meta data. In our solution, we establish that binding by encrypting meta data and decryption key in one blob.

While solving these problems, we also face the task of integrating our solution in the system described in section 2. That means that we want to preserve the property of simple file sharing. We also analyze what limitations concerning the authenticity of files exist in the cloud storage use case.

Contribution

We provide a broad analysis of token properties concerning security and usability. We compare available tokens and show to what extent they provide security and usability.

Afterwards, we provide a protocol in which the user controls the decryption operations on the token. The meta data of the files do be decrypted is shown to Alice before decryption. She can then confirm or abort the decryption.

We also show a general limitation in the setup including a token. When Alice's PC is not trustworthy, files cannot be shown authentically on it anymore. We can just prevent the attacker from adding new files, but this cuts down usability. We present and discuss this trade-off.

Our proof of concept implementation proves, that our system works in practice.

The results achieved here are independent of the cloud storage case. They might also be relevant for normal file system encryption.

3.2 Related work

As we have seen in chapter 3.1, giving the user control over the decryption operations increases security in the setup with a token. There are existing publications in the field of controlling transactions on a token, of course. In this chapter, we integrate our work with existing publications.

First, we describe how transactions are controlled on the new German ID, called neuer Personalausweis (nPA). We will conclude that the existing system's software might be changed to satisfy our purposes. Anyhow, changing this system is not easy, as there are no interfaces defined for our use case. We would have to misuse the existing system.

We then describe how transactions are controlled on a bank card. In this procedure, a Transaction Authentication Number (TAN) is generated to authenticate transactions on the bank account. Anyhow, this procedure is not used for decryption. It is used for authentication of the user at the bank.

Controlling transactions on the new German ID

Controlling transactions to be done one a smartcard is also a subject in the context of the nPA. In [32, pages 24 - 25], respectively [33, pages 26 - 27], controlling a transaction on the nPA is described. In the transaction (which is part of the "elektronischer Identitätsnachweis (eID)" procedure), personal data from the nPA shall be read. Our user Alice can control the transaction by choosing what kind of data may be extracted. She is able to abort the transaction in case it is not an expected one. In the references ([32] and [33]) is described that the information is shown to the user by the local software.

Anyhow, the information can also be shown on the trustworthy smartcard reader. Having a trusted display is a requirement for this approach[43]. The approach is described in [34, page 9, "Security of display module" and pages 17 - 19]. The key advantage is that the information about the transaction is shown on a device independent of the operating system of the PC. That means we can be sure that the information displayed is correct, even if we do not trust the operating system of the PC. Correct information means that what is shown actually happens on the smartcard.

If the information would be shown by the local software, our attacker Mallet, who controls the software, could manipulate the displayed information. He could show a transaction with very little data extraction and in reality extract all possible data (or, to give another example, fake the service provider).

As we can see in this example, displaying meta data about a transaction and getting user confirmation before doing the transaction has already been realized in the context of the nPA.

The meta data must be transferred from Alice's PC to her smartcard reader. This

[43]For example, cyberJack RFID komfort, see `https://www.reiner-sct.com/produkte/chipkartenleser/cyberJack_RFID_komfort.html` and cyberJack wave, see `https://www.reiner-sct.com/produkte/chipkartenleser/cyberjack_wave.html`

can be done in two ways. The first way is to use the reader's driver. Most drivers are closed source, but provide some kind of interface to transfer the information. The second way is to use pseudo-Application Protocol Data Units (APDUs) (see [34, pages 33 - 34]). We could try to encode our meta data as transaction information for the eID procedure and display it on the reader. This way, it might be possible to encode our encrypted file meta data as an input to a transaction on an nPA. Following this approach, we could use the already existing nPA system to realize our goals.

Anyhow, the existing system does not define any interfaces for our use case. We would have to misuse the existing interfaces. This is probably an overly complicated way to implement our idea.

Controlling transactions on a bank card

Another example of displaying meta data before performing a cryptographic operation with a private key is described in [35, pages 37 - 38]. This procedure is about generating TANs, e.g. to authenticate bank account transactions. In the bank account transactions use case, our user Alice has a smartcard, a smartcard reader with key pad and display and a PC. To generate a TAN, the website of Alice's bank displays a bar code containing the transaction's data on the computer's display. Alice then scans the bar code with the smartcard reader. The smartcard reader shows the transaction's data on its display and lets Alice confirm the transaction. It also lets her enter the PIN to unlock the smartcard. The TAN then is generated via the smartcard and displayed on the reader's display. Alice can enter it on the computer to do the transaction on her bank account. So in the procedure described here, transaction data is also confirmed on a trusted device.

This is very similar to what we want to do. We could also encode the meta data and the ciphertext as a bar code and use a reader with a bar code scanner. Anyhow, if the user always has to scan a bar code before decrypting a file, this cuts down usability. So in our use case, the approach taken here is not a solution.

Displaying meta data on the token

Another approach is to show the meta data directly on the token. We discuss the pros and cons of various tokens in chapter 3.3. One type of tokens discussed are smartphones. These provide great usability but also have a lot of channels available for attacks. A security analysis of using smartphones as tokens can be found in [36]. Using a smartphone as a token when decrypting or signing e-mails has been realized in [37][44]. This also includes showing information about decryption/signature operations and getting the user confirmation [37, pages 20 - 21, 23].

[44]Software available on `https://github.com/eriknellessen/Virtual-Keycard`

Conclusions

We showed that getting the user confirmation before performing a transaction is already implemented in existing systems. As a first example, we presented the eID functionality of the nPA. We showed that this existing system could be changed to implement our idea. On the other hand, due to the fact that no interfaces are provided for changing the system in the way we want, we would have to misuse the system. As this means much effort, we decided not to take this approach.

After that, we presented how bank transactions are controlled when using a bank card and the ChipTAN procedure. We concluded that in this procedure, transaction meta data is also confirmed before doing the transaction. On the other hand, this procedure is used for authentication, not for decryption. We did not see a possibility to change it to our needs.

Lastly, we described the possibility to display meta data directly on the token. Such a token can e.g. be a smartphone. We showed that displaying data of a cryptographic operation on a smartphone before doing the operation has in general already been implemented. We still need to make sure that the displayed meta data is bound to the decryption key, which shall be decrypted.

3.3 Design goals and abstract view

As we have seen in chapter 3.1, adding a token to our system can increase security and usability. By giving the user control over the cryptographic operations on the token, we can increase the security gain. We want to do that by showing meta data of the cryptographic operation on a trusted display. In chapter 3.2, we concluded that a smartphone can be used to do that. In this chapter, we propose a system which solves the stated problems. We describe it in an abstract way. In chapter 3.4, we also provide an implementation.

There are a lot of tokens available at the moment. In the first segment of this chapter, we define requirements a token should satisfy in our use case. In the following segment, we analyze to what extend existing tokens satisfy these requirements. No token satisfied all of our requirements.

We integrate the token into the existing system described in section 2. To give the user control over the decryption operations of single files, it must be possible to decrypt single files in this system. This is one requirement the cloud storage system has to satisfy in order to work with our idea in this part of the work. In the third segment, we analyze in detail, what requirements the system has to satisfy.

When the meta data is shown on the trusted display, we must guarantee that only a file with the displayed meta data is decrypted if the user confirms the transaction. In the fourth segment of this chapter, we solve this problem. We describe a protocol to bind a decryption key to meta data.

Lastly, we analyze whether sharing files/folders is still possible in the system we designed. We conclude that it can still be done in a simple way.

Requirements concerning token and reader

In this segment, we define requirements a token has to satisfy in our use case. We then examine to what extend available tokens satisfy these requirements.

First of all, security is an important requirement. A secure token should make it really hard to extract the secrets stored on it. That means that they should be stored in a place that is inaccessible by software and very hard to access via physical means.

Another important requirement is usability. Only if the token is inexpensive, easy to set up and easy to use, users will accept it.

To show meta data about the decryption operation, we need a trusted display. The display must not be modifiable directly from Alice's PC system. The PC can just send the data which shall be displayed. "Trusted" means that exactly the displayed data is also transferred to the token. That means that even when Mallet controls Alice's PC, the information shown on the display can be trusted. Figure 40 shows the necessary trust when showing meta data on a trusted display. Figure 41 shows it for the case that the meta data is shown on the PC.

To empower the user to confirm or abort transactions, we need some way of trusted input. The user should have some kind of button which can be used to confirm and abort transactions. This button must be independent of the rest of the PC, i.e. cannot be triggered by the PC. Even when Mallet controls Alice's PC, the button still works reliably.

A token should also provide a way of direct public key verification. That means that other people can verify the public key directly from the token. This way, the public key can still be verified correctly in case the user's PC system is controlled by Mallet. A graphical representation of direct verification is given in figure 42, for indirect verification see figure 43. When we compare indirect verification to direct verification, we can see that Alice's computer now also has to be trusted. So now Bob also has to trust in a device with a lower trust level (also comparing to figure 39).

If we verify the public key from the user's PC, the following attack is possible: Let us assume that Alice wants to share a folder with Bob. Let us also assume that Alice tries to get K_B^p from a server in the internet controlled by Mallet and that Mallet controls Bob's system (without the token). Mallet creates a key pair, of which we denote the attacker's public key (K_M^p). When Alice asks for K_B^p, Mallet provides K_M^p. After getting K_M^p, Alice verifies it using Bob's system. As Mallet controls Bob's system, it shows information about K_M^p when asked for information about K_B^p. The files are then encrypted with K_M^p on Alice's system. Mallet has access to the shared encrypted files on Bob's system.

So when Alice shares the folder, Mallet can read the files in plain, without having Bob confirm a transaction concerning that folder on the token. As Bob's system is under Mallet's control, it is no problem to re-encrypt the folder with the correct public key from the token, and present the resulting ciphertext to the token when Bob wants to decrypt the folder. So neither Alice nor Bob are able to detect this attack without direct public key verification.

Anyhow, direct public key verification is usually possible on every kind of token.

Some tokens can communicate with a trusted display and can display the fingerprint of the public key/a QR code encoding it. In any case, direct public key verification can be done by handing the token over to the person who wants to verify the public key. But this can create new problems concerning security. There are tokens without an additional barrier for performing operations on them. To perform operations, only having the token is already enough (without knowing an additional secret like a PIN or having the correct fingerprint etc., e.g. Universal Second Factor (U2F) on a Yubikey[45]). These tokens do not provide a good level of security, as stealing them means being able to use the secrets residing on them. Tokens protected with an additional secret should receive that secret in a trusted environment. This could for example be a trusted keyboard. Otherwise, we need to trust the user's software and hardware when unlocking the token.

To show why trusting the user's software and hardware when unlocking the token is a problem, we describe an attack assuming this approach. Let us assume Alice takes this approach with her token and Mallet controls her system (without the token). When Alice enters her PIN in the system controlled by Mallet, Mallet knows the PIN. Stealing Alice's token is now enough to decrypt every file Mallet wants to. But stealing is maybe not even required. As Alice wants other people to verify her public key, she might give her token to other people (this does not directly imply a security risk, as it is protected by a PIN). Anyhow, if Mallet already knows the PIN and then gets the token for public key verification, he is able to decrypt anything he wants.

So to summarize the last two ideas, we present two secure scenarios to directly verify public keys. In the first scenario, the token is able to display the public key fingerprint in a trusted environment. The token does not need to be handed over to the verifier. To give an example, a smartphone can show a QR code of the public key's fingerprint on its display. In the second scenario, we have to hand the token over. Then, the token needs to be sufficiently protected against misuse. That means that we need a trusted keyboard. Having the trusted keyboard is a good idea anyhow, as it prevents Mallet from getting to know our PIN, even when he controls our PC. That means that only stealing the token is not enough for him to perform a successful attack.

Summarized, our requirements (concerning the whole trusted token and reader system) are:

- High level of security
- Good usability
- Trusted display
- Secure confirm/abort button
- Secured by a secret (e.g. a PIN)
- Secure input method for the secret (e.g. PIN keyboard)

[45]See `https://www.yubico.com/products/yubikey-hardware/`

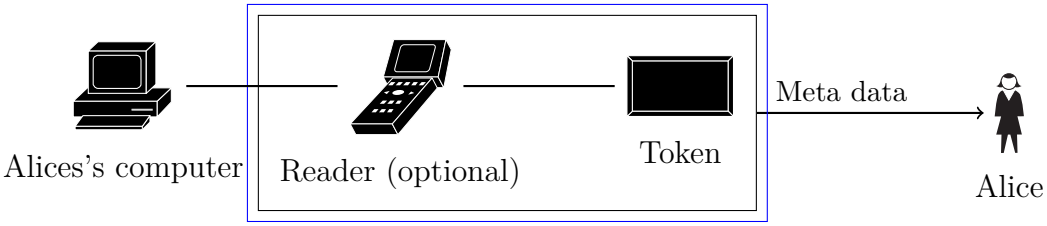

Figure 40: Meta data shown on token or reader, trusted devices are framed in blue. Token and reader are framed in black to show that the meta data/confirmation request can come from each of them.

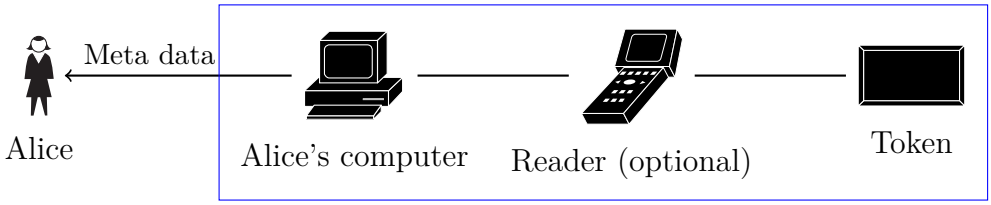

Figure 41: Meta data shown on computer, trusted devices are framed in blue.

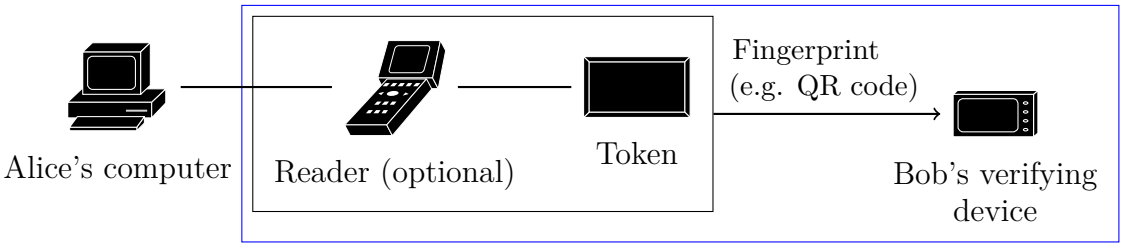

Figure 42: Direct verification, trusted devices are framed in blue. Token and reader are framed in black to show that the verification information can come from each of them. The reader can also be part of Bob's verifying device or system.

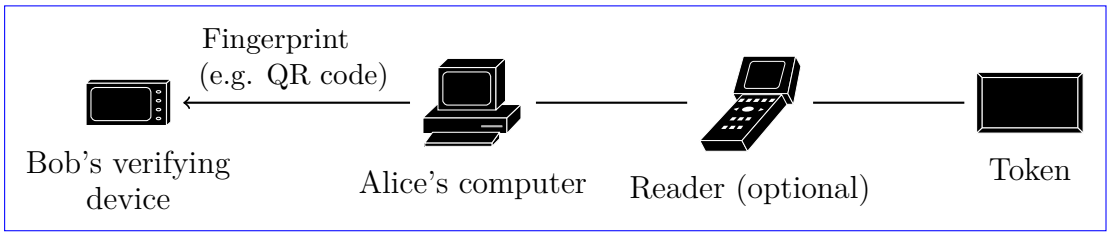

Figure 43: Indirect verification, trusted devices are framed in blue.

Comparison of available tokens

There are a lot of devices that can be used as a token. We discuss the pros and cons of smartphones, smartcards and YubiKeys in our use case. Our results are summarized in figure 44.

A smartphone has a lot of possibilities to communicate with other devices, e.g. network, bluetooth, infrared or Near Field Communication (NFC). That means that there are a lot of channels that can be used by attackers. To at least store private keys securely, Trusted Platform Modules (TPMs) could be used (but are usually not used at the moment). So our security requirement for the private keys is not satisfied.

On the other hand, the various possibilities of communication mean that smartphones are capable of being integrated into a lot of systems (and that additional hardware, like a smartcard reader, is not always needed). Smartphones are widely spread and very easy to use. That means that supporting smartphones as tokens enables a lot of users to use the system. The usability requirement is satisfied.

We can use the smartphone's display to show the meta data independent of the PC system. The requirement of a trusted display is satisfied. The user confirmation is also done directly on the display. So the requirement of a secure confirm/abort button is satisfied.

Of course, the smartphone can be secured by e.g. a PIN. The PIN can be typed in on a trusted keyboard of a trusted reader or directly on the smartphone's display.

A smartphone having a trusted display and secure input might seem irritating at first sight, because a smartphone does not provide a good level of security (as already stated). Anyhow, our attacker model assumes that the attacker controls Alice's PC, but not her token (see chapter 3.1). As long as the smartphone is not controlled, its display/input is trustworthy. Of course, a smartphone does not provide a good security level and can be infected much easier (e.g. when connected to the infected PC via Universal Serial Bus (USB)) than e.g. a smartcard. We do not recommend using smartphones as tokens in production environments. Anyhow, disregarding the security properties, a smartphone has all the other advantages we need in our use case. Additionally, it provides an NFC interface. That is why it can easily be integrated as a token and provides a comfortable way to implement a proof of concept. This proof of concept could be enhanced in security by providing a more secure token, which also satisfies all other requirements, in the future.

Smartcards[46] have been developed specifically as secure storages for private keys. By making the private keys inaccessible by software and very hard to access via physical means, they satisfy our security requirement (for the private keys).

Smartcards usually offer an interface using contacts and/or a contactless interface. They are also available with a bluetooth interface. In the first two cases, a smartcard reader is needed to communicate with a smartcard. A smartcard reader can be included in smartphones. Anyhow, if users want to use a reader directly from their PC, they have to buy additional hardware. Besides, the setup is more complicated than using a token without an additional reader. It is hard to set up and use for the user. Anywhere

[46]See e.g. `https://g10code.com/p-card.html`

the smartcard shall be used, a smartcard reader must be available. For these reasons, the usability requirement is not satisfied.

There are smartcard readers with a trusted display available. If we use one of these, our trusted display requirement is satisfied. These readers usually also provide a trusted confirm/abort button and a trusted PIN keyboard. A smartcard is usually protected by a PIN. So all of the remaining requirements are satisfied, too.

A display card satisfies just the same requirements as a smartcard. It just provides more security, if no appropriate smartcard reader is available. This is because it has a trusted display already included.

A Yubikey includes a smartcard. That is why it is very hard to extract private keys. The security requirement for the private keys is satisfied.

A Yubikey can be used via USB. As no additional reader is used, user acceptance is given in our use case. The usability requirement is satisfied. The better Yubikeys also provide NFC.

A Yubikey can be regarded as a smartcard including a reader. Anyhow, the "included reader" does not satisfy our requirements. It does not include a trusted display. Nevertheless, the Yubikeys supporting NFC might be accessible via readers with a trusted display (which then would mean low usability). Summarized, our trusted display requirement is not satisfied if we assume good usability.

Most Yubikeys include a button to confirm transactions. This button is used to authorize U2F transactions. It is not used to authorize decryption operations on the included OpenPGP applet. That is why our secure button requirement is not satisfied.

The OpenPGP applet on a Yubikey is protected by a PIN. Anyhow, there are other functions on the token, that are not protected by a PIN, e.g. the U2F function. To perform an attack on U2F, also the user password is required. Anyhow, if the attacker Mallet controls the victim's PC, he already has it. It is not safe to give the Yubikey to someone else to verify the public key. So our PIN requirement is not satisfied.

Yubikeys do not provide a method to enter the PIN on a trusted keyboard. The Yubikeys supporting NFC might be accessed by a reader with a trusted keyboard, but this would again cut down usability. So assuming good usability, our secure PIN input requirement is not satisfied.

Nitrokey[47] sells products similar to Yubikeys. They do not own a secure button at all. The other properties are very similar to those of the Yubikeys.

The comparison of tokens is summarized in figure 44. We can see that none of the tokens satisfies all of our requirements. A token satisfying all requirements could be a smartcard containing a trusted display and keyboard[48]. Our analysis motivates the development of such cards.

[47] `https://www.nitrokey.com/`

[48] E.g., `http://www.maestrocard.com/de/privatkunden/innovationen_technik/displaycards.html`

Token	Security for private keys	Usability	Trusted display	Secure button	Additional secret	Secure input method
Smartphone	Bad	Good	Yes	Yes	Yes	Yes
Smartcard	Good	Bad	Yes (in smartcard reader)	Yes (in smartcard reader)	Yes	Yes (in smartcard reader)
YubiKey	Good	Good	No	No (not for decryption)	Not for U2F	No

Figure 44: Comparison of available tokens.

Requirements concerning the "File encryption/decryption system"

In the last two segments, we analyzed, what requirements a token has to satisfy in our use case and to what extend these requirements are satisfied by available tokens. But not only tokens have to satisfy requirements. We integrate the token into the existing system from section 2. To make this possible, the system has to meet certain requirements. In this segment, we show these requirements. As we will see, the system implemented in section 2 satisfies them.

Tokens usually do not provide much computation power. That means that they are not able to perform large cryptographic operations. That is why hybrid encryption should be used to encrypt files. Hybrid encryption means that symmetric encryption is used (on the PC) to encrypt the file and only the symmetric key is encrypted via the asymmetric cryptographic system. When decrypting, only the symmetric key has to be decrypted in the asymmetric cryptographic system. Afterwards, the symmetric cryptographic procedure can be used. As this is much better for performance, it is usually done like this, also when not using a token. In the implementation of our system, this is also the case.

To increase security, we want to show meta data before decryption on a trusted display. To do so, the cloud storage system must support single file decryption. That means that it is possible to enable on the token the decryption of single files on the PC without enabling the PC to decrypt other files. As we discuss in the following segment, there is a trade-off between usability and security here. If we want to implement a reasonable trade-off, we can enable the decryption folder-wise. The system from section 2 can easily be changed to support this. We just have to remove the property of possible recursive decryption.

In this segment we showed the requirements the system from section 2 has to satisfy, so our token can be integrated. We concluded that hybrid decryption and folder-wise decryption are required. Hybrid decryption is already provided, while folder-wise decryption can easily be implemented by changing the existing system.

Binding meta data to a ciphertext

As we have already seen, giving the user control over the decryption operations on the token increases security. To do so, we show the meta data on a trusted display and ask for user confirmation. But we still need to make sure that the decryption operation only enables the PC to decrypt the file with the displayed meta data. We have to bind the meta data to the decryption key of the file with the shown meta data. This is the problem we solve in this segment. We do so by encrypting meta data and decryption key in one blob.

If our user Alice has to confirm every file decryption, she has to do a lot of work. It would be more comfortable if she would not have to do any confirmation at all. But then, there also is no gain in security. We could also confirm only one decryption and make everything decipherable from one key as in section 2. But then again, we would not gain security. Confirming one decryption is just another way of unlocking the token (like a PIN). We could also decrypt folders instead of files. Assuming our user Alice uses folders to structure her data, this already means a gain in security. On the other hand, only confirming the decryption of every folder can be a big relief for the user (e.g. when viewing a folder full of photos). So assuming the files are structured in folders, this is a good trade-off between security and comfort. Confirming the decryption of every single file is the securest and the least comfortable solution.

Before decrypting the symmetric key, we present meta data to the user. So the PC has to transfer the encrypted symmetric key and the meta data to the token. We need to verify the meta data's binding to the key. If we would not do this, the following attack would be possible. Let us assume that the attacker Mallet controls Alice's system and wants to decrypt file f. At some point, Alice wants to access her file f', so she unlocks the token. Mallet then transfers the encrypted symmetric key for decrypting f and the meta data for f' to the token. From Alice's point of view, everything is working as expected, so she confirms the decryption. Mallet is then able to decrypt f.

One way to bind the meta data to the key is to encrypt meta data and plain encryption key as one blob, or to encrypt the meta data's hash value and the plain encryption key as one blob. Encrypting the hash value has the advantage that the decryption operation on the token has to be done on data of a fixed and small length (which is small enough, so that only one decryption operation is needed). Another advantage is that the meta can already be shown before decryption/PIN input, but still has to be transferred to the token only once. A disadvantage is that the meta data is visible to the provider, as it has to be saved as plaintext.

If we follow this approach, our attacker Mallet is able to create (meta data, decryption key) pairs. But this is not important to us, as he is not able to combine another meta data with a specific encrypted key. To do this, Mallet would have to solve the following problem. We presume that we use RSA as asymmetric cryptographic system. Let us assume that Mallet wants to decrypt a file f' with the meta data m' and the decryption key d' (d' is not known to Mallet, while $(m'|d')^e \bmod n$ is, e and n being the exponent/modulus from the RSA procedure). We also assume that Alice wants to decrypt the file f with the meta data m and the decryption key d. To perform the attack,

Mallet has to find the ciphertext $(m|d')^e \bmod n$ knowing m', m, e and n (and maybe also d from a previous decryption procedure). There is no way known to us to efficiently solve this problem without knowing the private key. In asymmetric cryptographic systems, the attacker Mallet is always able to create (plaintext, ciphertext) pairs as desired. In a secure cryptographic system, this should not let him gain any knowledge. This might be a first hint that the presented problem might also be hard to solve. In practice, RSA is usually done with a randomized padding. This could make the problem even more difficult.

So if the attacker Mallet is not able to solve the described problem efficiently, our system is secure. But even if that assumption is not valid, we can make our system secure. By signing meta data (and letting the user confirm the signature operations), Mallet cannot create the new (meta data, decryption key) pair $(m|d')^e$ authentically. In the following paragraph, we describe this procedure. We also explain that the procedure prevents another attack.

If we sign the meta data, we are able to only accept meta data which has been signed by a trusted key. So the attacker Mallet can only transfer meta data to the token, which has been signed by a key which we trust in. It is not possible for him to transfer any kind of desired meta data. Anyhow, this means that we have to ask Alice for permission when creating a signature on the token. We also have to show the meta data to be signed. If we would not do that, Mallet could just create a signature for the desired meta data.

So at this point, we have a trade-off in security and usability. If we do not sign the meta data, the following attack is possible. Let us assume that the attacker Mallet controls Alice's PC. He creates a new encrypted file f and a corresponding meta data and encryption key. For example, the file f could state that it comes from Bob (e.g. it is a letter ending with "Yours sincerely, Bob"), but contains misleading information (like "Meeting Saturday, 13:37.", when in reality, the meeting is on Friday). If we do not sign the meta data, Alice is not able to detect, that f does not come from Bob.

Anyhow, if we sign the meta data, the attacker could still use meta data of any of Bob's existing files. On the token, nothing unusual would be detectable. On the PC, the attacker could show a modified version of the file, again containing the misleading information.

So confirming every signature on the token, which is a cutback of comfort, leads to a slightly more secure system. Slightly more secure means, that Mallet can only use existing meta data and is not able to create new authentic meta data. Of course, we should still sign the meta data and verify signatures on the PC. If we would not do this, we would enable the cloud storage provider to forge files very easily, as we do not verify their authenticity in any way.

The procedure of encrypting (and optionally signing) an encryption key for a file/folder is shown in figures 45 (using a smartcard as token) and 46 (using a smartphone as token).

When decrypting an encryption key, the decrypted meta data is shown on the token/reader. If we use the hash value instead, the decrypted hash value is compared to the (shown and confirmed) meta data's hash value. If it is the same, the decryption

key is transferred to the PC, else the operation is aborted.

The procedure of decrypting (and optionally verifying a signature of) an encryption key for a file/folder (using the hash value method) is shown in figures 47 (using a smartcard as token) and 48 (using a smartphone as token).

In this segment, we presented a solution to the problem of binding meta data to a decryption key. The problem arose when we described that we wanted to show meta data before the decryption operations on a trusted display. We needed to make sure that no existing files with a meta data different to the shown one could be decrypted on the PC. We solved this problem by encrypting meta data (respectively its hash value) and the decryption key in one blob. We also explained limitations concerning authenticity when the PC is in the control of the attacker.

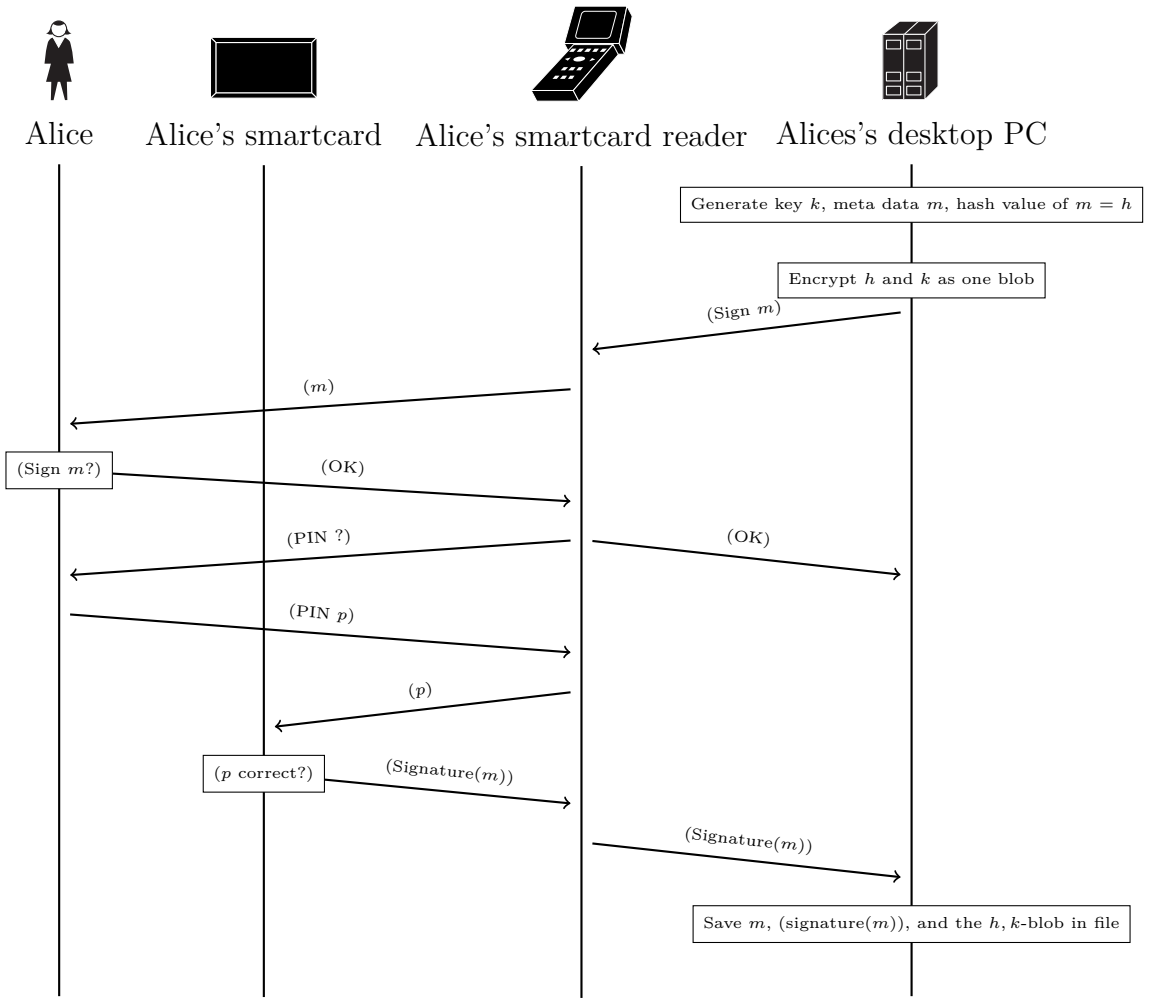

Figure 45: Alice encrypts a file/folder locally and optionally establishes authenticity using her smartcard.

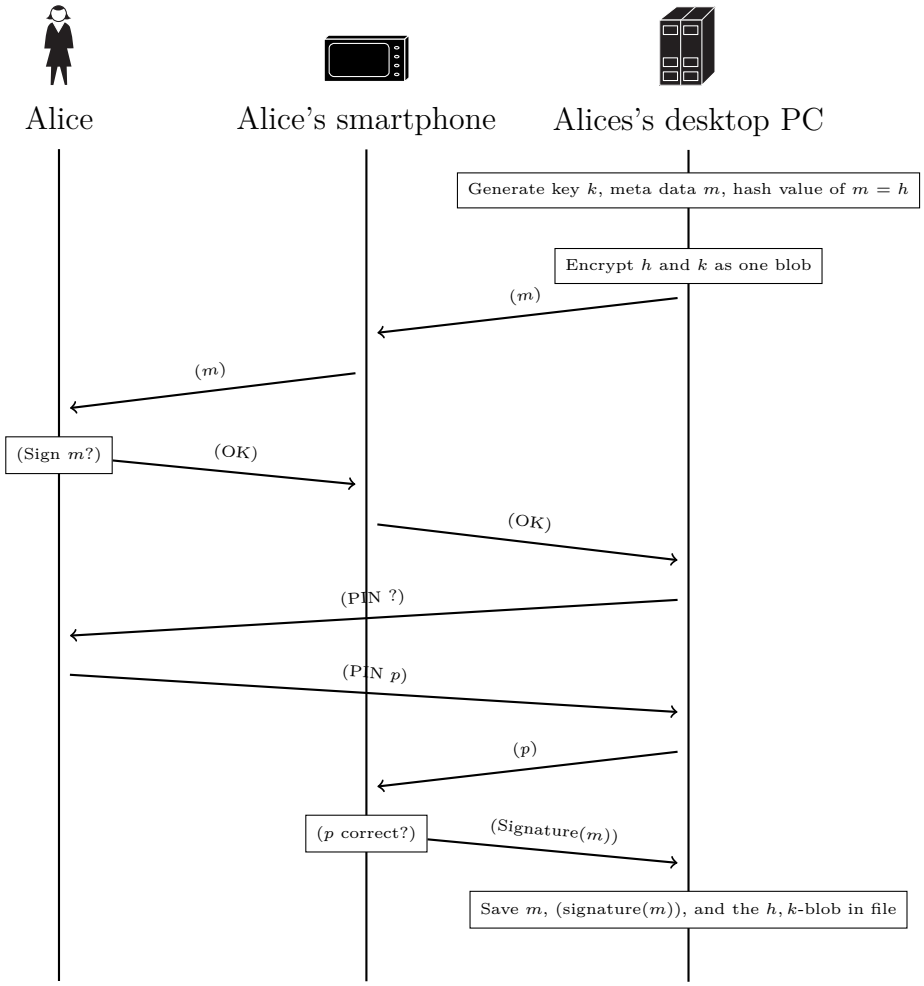

Figure 46: Alice encrypts a file/folder locally and optionally establishes authenticity using her smartphone. It would also be possible to integrate the smartphone as a smartcard using NFC and a reader. Another possibility would be to let the user enter the PIN directly on the smartphone to increase security.

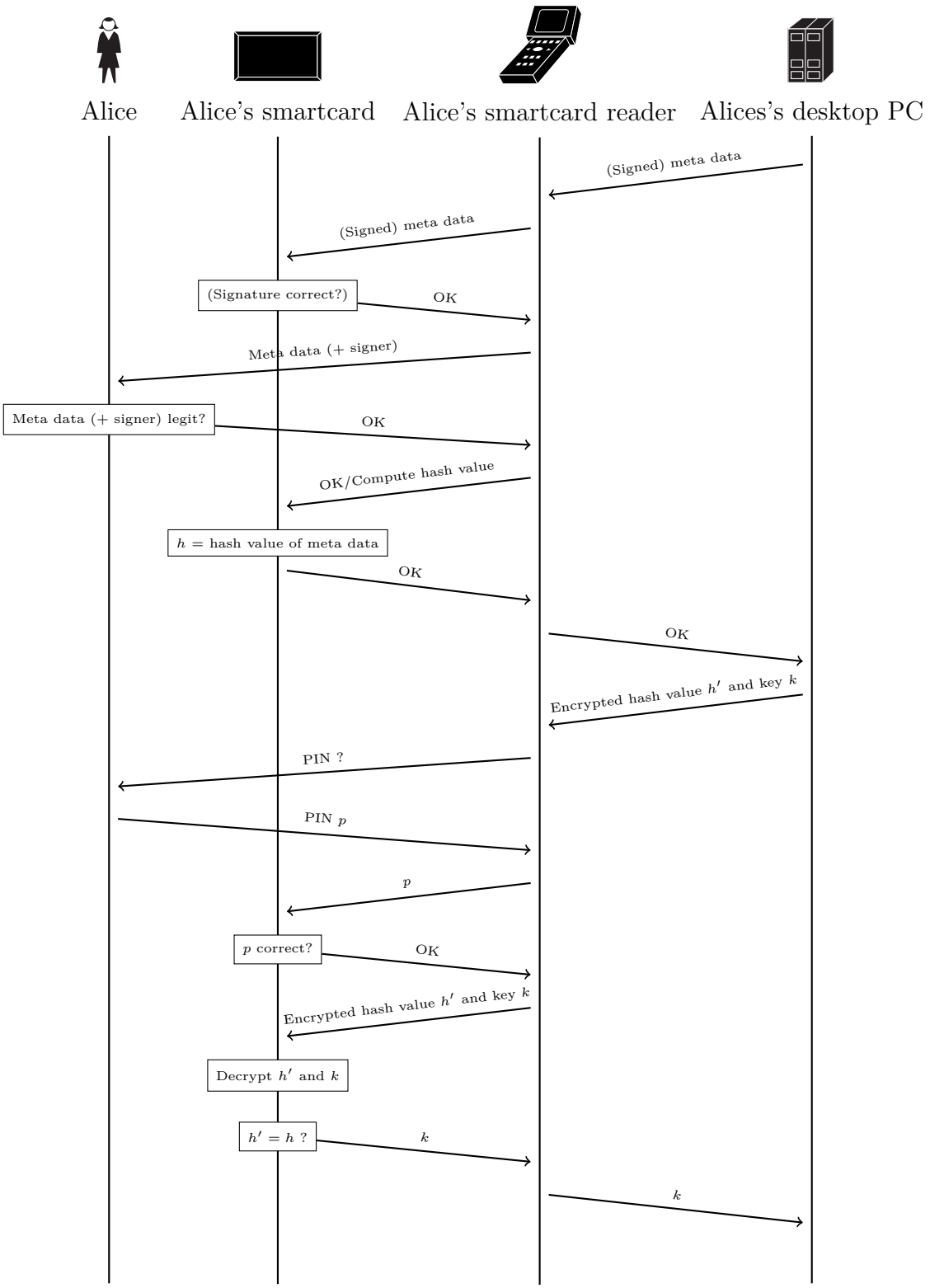

Figure 47: Alice decrypts a file/folder using her smartcard. The signature verification and the hash value comparison could also be done on the reader.

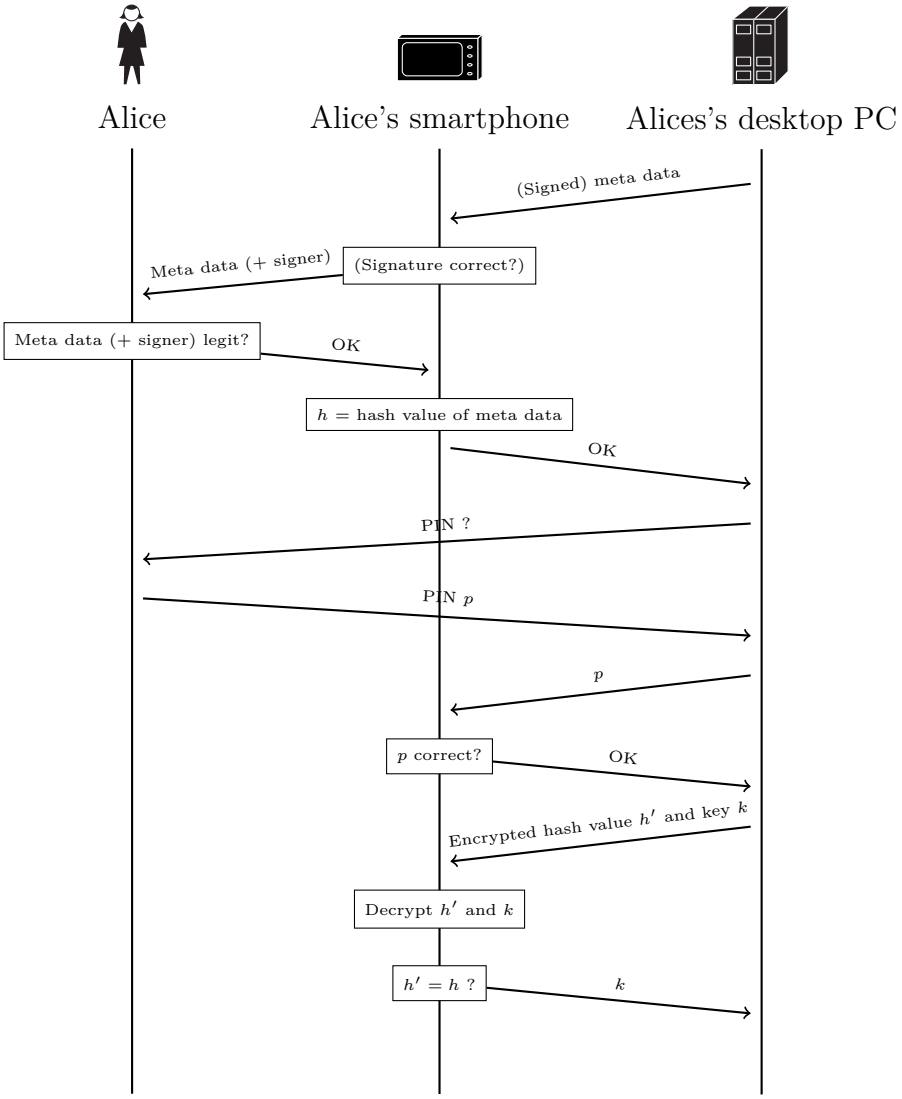

Figure 48: Alice decrypts a file/folder using her smartphone. It would also be possible to integrate the smartphone as a smartcard using NFC and a reader. Another possibility would be to let the user enter the PIN directly on the smartphone to increase security.

Sharing files/folders

In the last segment, we proposed a system to bind meta data to a decryption key. We showed protocols which describe in what way the files are encrypted/decrypted and authenticity is established/verified. We now describe how it is still possible to share files/folders when following the described protocols.

Sharing a file/folder with Bob works just like encrypting a folder for later decryption on the same PC. We just substitute our public key with Bob's. On our PC, we have all the information Bob needs to do the same procedure on his PC. So in the end, from a cryptographic point of view, the only difference between the saved files (referring to figures 45 and 46) is the signature being done with Alice's private key instead of Bob's.

To decrypt the files/folder, Bob can proceed just as shown in figures 47 and 48.

Verifying authenticity of shared files/folders is possible as already described in the last segment. It makes sense to do it on the PC because then, the cloud storage provider cannot forge messages easily. If we assume that our PC is controlled by the attacker Mallet, verifying only makes sense on the token. We can then prevent Mallet from creating new files, but not from changing already existing ones. We pay for that by confirming every signature operation.

Conclusions

In this chapter, we presented our system design. Our goal was to show meta data on a trusted display (of a token or reader) before performing a decryption operation. This way, the security of the system can be increased.

In the first segment, we examined, what requirements a token has to fulfill in our use case. We concluded by providing a list of six requirements. Afterwards, we presented available tokens and analyzed to what extend they satisfy the six requirements. We concluded that none of the available tokens satisfies all of the requirements. We showed, however, that a smartcard with an included trusted display and PIN keyboard would be perfect for our use case.

We then described the problem of binding meta data to a decryption key. Doing so is necessary to make sure the information shown on the display is trustworthy. Trustworthy means that if we confirm the operation, only a file with the matching meta data can be decrypted and none of the other files we encrypted when our system was not controlled by an attacker. We first showed what requirements the system described in section 2 has to satisfy to make this possible. We concluded that hybrid decryption and folder-wise decryption are required. Hybrid decryption is already done in the system, the folder-wise decryption could easily be implemented by doing small changes.

Afterwards, we proposed a system to establish the binding between meta data and decryption key. Our idea is to encrypt meta data and decryption key in one blob. We also analyzed, what limitations concerning authenticity exist in general, assuming that the PC is controlled by Mallet. We concluded that we can prevent Mallet from creating new authentic files by paying with usability drawbacks.

Lastly, we showed that sharing files/folder is still easily possible in the proposed system.

3.4 Implementation

In the last chapter, we described our system design from an abstract point of view. We now provide a proof of concept implementation[38] (a tutorial on how to use the software can be found in the Readme in the appendix of this work). We describe how we implemented the system described in chapter 3.3 and list the existing software we used.

For our proof of concept, we use a smartphone as a prototype token. A smartphone does not provide a good level of security, but has interfaces to develop a proof of concept comfortably. The procedure for encrypting a file encryption/decryption key is shown in figure 46, the decryption procedure in figure 48. We chose usability over the small security gain and did not implement the signature verification on the token.

To implement the encryption procedure, we used libgcrypt[49]. To receive the public key from GnuPG, we used GPGME and python-pgpdump[50]. Libgcrypt was also used to calculate the hash value.

To implement the decryption procedure, we used OpenSC[51] to access the token. We were able to use the included OpenPGP card driver. To receive the PIN from the user, we used pinentry[52].

To use an Android smartphone as a smartcard, we used Host-based Card Emulation (HCE)[53].

As we used GnuPG on the PC, we had to implement a correspondent smartcard functionality on the smartphone. GnuPG has a standard for tokens, see [39]. There is also a Java Card Applet available implementing that standard[54]. This project has been forked by Yubico[55]. The standard does not comply[56] with the Public Key Cryptography Standard (PKCS)#11 standard [40]. But there is a program available to use PKCS#11 instead of the OpenPGP smartcard standard[57]. Using this program, we could use a PKCS#11 compliant smartcard with GnuPG instead of a smartcard compliant to [39].

We used jCardSim[58] to provide the Java Card[59] functionality, which is needed by the Java Card Applet. We then installed the Yubico Neo OpenPGP applet on the virtual smartcard created by jCardSim.

[49] https://gnupg.org/related_software/libgcrypt/index.html
[50] https://github.com/toofishes/python-pgpdump
[51] https://github.com/OpenSC/OpenSC
[52] https://www.gnupg.org/related_software/pinentry/index.de.html
[53] https://developer.android.com/guide/topics/connectivity/nfc/hce.html
[54] https://sourceforge.net/projects/javacardopenpgp/
[55] https://github.com/Yubico/ykneo-openpgp
[56] See e.g. https://lists.gnupg.org/pipermail/gnupg-devel/2005-September/022362.html
[57] http://gnupg-pkcs11.sourceforge.net/man.html
[58] https://github.com/licel/jcardsim
[59] http://www.oracle.com/technetwork/java/embedded/javacard/overview/index.html

One remaining problem is that jCardSim does not support persistent saving of a simulator's state yet. But the project identified this problem and is working on it[60]. For the meantime, an existing solution is described in [37, pages 32 and following].

With this setup, we emulate a normal OpenPGP card, but we can also intervene in the communication. We can work on data before sending it to the applet, and we can implement additional functionality in the Android app. We used this possibility to implement the protocol described in chapter 3.3. As a hash function, we used SHA-256 according to [24]. A screenshot of the implemented Android App is given in figure 49.

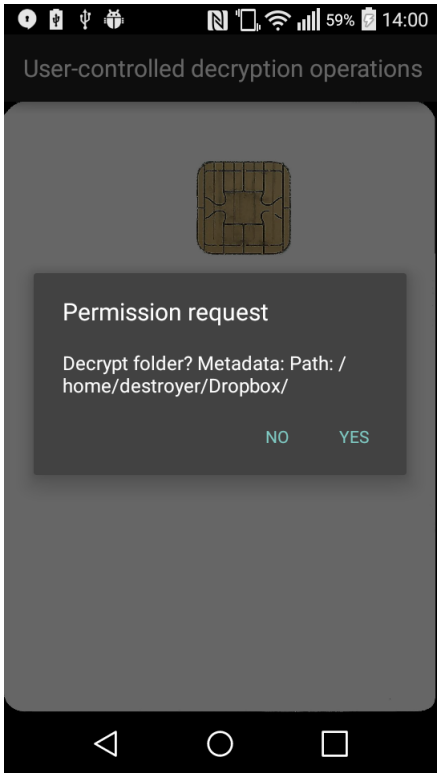

Figure 49: Screenshot of the implemented Android App when demanding user confirmation.

In this chapter, we provided our proof of concept implementation. We used a smartphone as a token. By using HCE, we emulated a smartcard. We then used jCardSim to create a virtual Java Card on the smartphone. By installing the Yubico Neo OpenPGP applet on the virtual smartcard, we provided the OpenPGP card functionality needed to communicate with GnuPG on the PC. An overview of the used software is given in figure 50.

[60] `https://github.com/licel/jcardsim/issues/77`

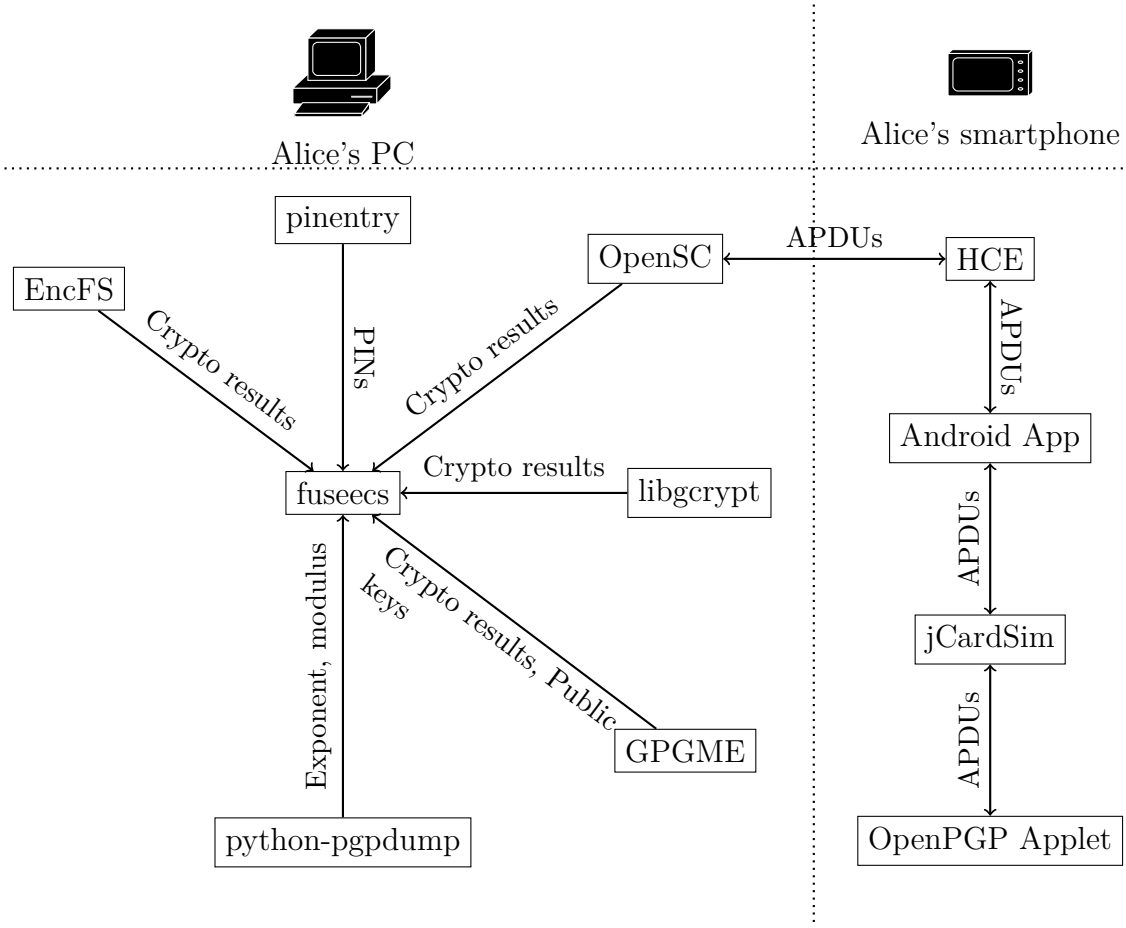

Figure 50: Involved software when using a smartphone as a token.

3.5 Conclusions

In this part, we carved out the possible security advantages when using a token. We proposed a system to achieve these security advantages. We now present our conclusions to clearly show the results of this part of the work.

To show why it is necessary to treat the subject of showing meta data before performing decryption operations, we showed that using a token only protects the confidentiality of Alice's data until she uses her token for the next time. In contrast, controlling the decryption operations on her token empowers her to protect one specific data's confidentiality until she uses that specific data for the next time.

We then analyzed existing work concerning similar systems. An existing procedure doing similar things is the eID function of the nPA. We described in what way this system contains controlling operations on a token. We also explained abstractly how this existing system could be changed to implement our idea.

We then described, what demands we have concerning a token to be used in our

system. While doing this, we described possible attacks in case our requirements are not satisfied. We presented existing tokens and explained to what extend they satisfy our requirements.

Afterwards, we proposed a system resisting the described attacks. It binds the meta data to the encryption/decryption key by encrypting the meta data's hash value and the key in one blob. We also showed the authenticity limitations based on our assumptions about the attacker.

We then presented our open source implementation of our system. It consists of two parts. The first part is an Android app, the second part is a Linux program.

The Android app provides the functionality to use an Android smartphone as a token using NFC. When Alice decrypts her cloud storage data, the meta data about the files to decrypt is shown on her smartphone. She has to confirm it to do the decryption. This app uses the OpenPGP Java card applet and jCardSim.

The Linux program is a modification of the implementation from section 2. It has been modified to perform direct asymmetric encryption locally and direct asymmetric decryption on the token. For the communication with the token, OpenSC, pinentry and libassuan are used. For the direct asymmetric encryption, GPGME, python-pgpdump and libgcrypt are used.

The contribution in this part of the work was to present a system providing a higher level of security when using a token for decryption. The results are not specific for the cloud storage use case and could, for example, also be useful for general file encryption/decryption (like encryption of a whole hard disk drive).

4 Conclusions

As stated in the introduction (section 1), our goal was to provide a possibility to use cloud storage services without losing privacy.

In section 2, we showed that client-side encryption is a requirement, but not a sufficient one. We did so by describing a man-in-the-middle attack the cloud storage provider is able to perform. In the cloud storage applications we examined, this attack was possible. So in these cloud storage systems, Alice needs to trust her cloud storage provider not to spy on her.

We proposed a system that performs client-side encryption and makes the trust in the provider as a CA needless. Attacks as described in section 1 and chapter 2.1 are not possible anymore within our system. We replaced the trust in the provider with the trust in oneself. That means that Alice herself has to do the public key verification.

We described how this can be done comfortably. By using smartphones and QR codes for verification, Alice does not need to verify public key fingerprints by comparing byte by byte. She can just scan the QR code and let her software do the verification. Smartphones provide a high level of comfort because they are widely spread and most of the time at hand. So whenever two persons meet, they most probably have everything with them to do a comfortable public key verification.

We showed that encrypting cloud storages without changing the cloud storage application can be done transparently for the daily usage. Daily usage means creating new files, changing and deleting files and accessing them from several devices. When we use the full functionality, not everything can be done transparently. For example, Alice cannot use the following features without noticing that she uses an encrypting program:

- Sharing files/folders
- Accessing older file versions
- Accessing folders shared by Bob (Alice has to verify that she expects the folder to be signed by Bob)

So providing a basic cloud storage functionality and encrypting a cloud storage are separate problems for the daily routine. For the remaining features, the cloud storage applications could define ways to change their program, like add-ons. Another way would be to implement the features in the encrypting/decrypting program. Anyhow, this is not purely transparent, as Alice would notice that she does not use her original cloud storage application anymore. Additionally, this approach is hard to implement and depends on the cloud storage provider. If the cloud storage provider does not communicate the interfaces, we have to find out how they work. If the interfaces are changed, our program also has to be changed. Summarized, we presented a system, which is purely transparent for the daily usage. For the use cases which cannot be done purely transparent, we described ways to keep up a good level of comfort.

Our implementation proves that the system as proposed works in practice. It uses GnuPG for the certificate management and provides transparency for the daily usage

as described here. The comfortable public key verification is already possible by using OpenKeyChain. We described how it can be done using existing software.

In section 3, we added a token to the setup in section 2. Tokens are independent of the PC system. That is why a token can be used on any system without having to transfer the secret key to that system. Additionally, tokens can be protected by PINs (or other secrets). The token is locked after too many wrong PIN inputs. That is why a secure PIN can be way shorter than a secure password. So tokens can increase usability and security.

We analyzed in what way security is increased when a token is used. We assumed that the attacker Mallet controlled Alice's system. If Alice's system would be trustworthy, we would not increase security by adding a token. We concluded that, without further precautions, a token protects Alice's privacy, until she accesses her files for the next time. We concluded that we could achieve a higher level of security by controlling the decryption operations. That way, Alice's file f is not only protected until she accesses her data in general for the next time. It is protected until she accesses f for the next time.

We defined requirements for a token in that scenario. Then, we compared these requirements with the properties of available tokens. No token satisfied all of our requirements.

By outlining attacks, we showed that a binding between meta data and decryption key is necessary. We established that binding by encrypting the meta data's hash value and the decryption key in one blob. We proposed a protocol that shows the meta data on the token and gets Alice's confirmation before decryption. Before transmitting the result of the decryption to the PC, the binding between meta data and decryption key is checked. We described how this protocol can be applied to sharing files.

We also carved out general limitations. There is a trade-off between usability and security when determining granularity. Confirming a decryption for every single file cuts down usability, while being a very secure solution. We can increase usability while still providing more security than in the original scenario by confirming the decryption of folders.

Another limitation exists when checking authenticity of files. We cannot check the authenticity of whole files on the token, as files may be very large. We can check a file's meta data's authenticity. But then, the attacker can still show forged content on the PC system. The possible security gain is just to permit only existing meta data. But we have to pay for this gain by letting the user confirm signature operations, which cuts down usability.

Our implementation proves that the system as proposed works in practice. It is a modification of the system implemented in section 2. It uses a smartphone as a token and accesses it via an NFC reader.

So in the end, we showed ways to help people to use cloud storage services without losing their privacy. We showed how trusting in the provider can be overcome, how encryption can be done transparently, and how tokens can increase usability and security.

References

[1] Glenn Greenwald. *No place to hide: Edward Snowden, the NSA, and the US surveillance state*. Macmillan, 2014.

[2] Martin Mulazzani et al. "Dark Clouds on the Horizon: Using Cloud Storage as Attack Vector and Online Slack Space." In: *USENIX Security Symposium*. San Francisco, CA, USA. 2011, pp. 65–76.

[3] Duane C Wilson and Giuseppe Ateniese. "*To Share or not to Share* in Client-Side Encrypted Clouds". In: *Information Security*. Springer, 2014, pp. 401–412.

[4] Nikos Virvilis, Stelios Dritsas, and Dimitris Gritzalis. "Secure cloud storage: Available infrastructures and architectures review and evaluation". In: *Trust, Privacy and Security in Digital Business*. Springer, 2011, pp. 74–85.

[5] Chetan Bansal et al. "Keys to the cloud: Formal analysis and concrete attacks on encrypted web storage". In: *Principles of Security and Trust*. Springer, 2013, pp. 126–146.

[6] Karthikeyan Bhargavan and Antoine Delignat-Lavaud. "Web-based Attacks on Host-Proof Encrypted Storage." In: *WOOT*. 2012, pp. 97–104.

[7] Stephan Groß and Alexander Schill. "Towards user centric data governance and control in the cloud". In: *Open Problems in Network Security*. Springer, 2012, pp. 132–144.

[8] Marc Mosch. "User-controlled data sovereignty in the Cloud". In: *Proceedings of the PhD Symposium at the 9th IEEE European Conference on Web Services (ECOWS 2011), Lugano, Switzerland (September 2011)*.

[9] Dominik Grolimund et al. "Cryptree: A folder tree structure for cryptographic file systems". In: *Reliable Distributed Systems, 2006. SRDS'06. 25th IEEE Symposium on*. IEEE. 2006, pp. 189–198.

[10] Saman Zarandioon, Danfeng Daphne Yao, and Vinod Ganapathy. "K2C: Cryptographic cloud storage with lazy revocation and anonymous access". In: *Security and Privacy in Communication Networks*. Springer, 2012, pp. 59–76.

[11] Huijun Xiong et al. "Cloudseal: End-to-end content protection in cloud-based storage and delivery services". In: *Security and Privacy in Communication Networks*. Springer, 2012, pp. 491–500.

[12] Nesrine Kaaniche, Maryline Laurent, and Mohammed El Barbori. "CloudaSec: A Novel Public-key Based Framework to Handle Data Sharing Security in Clouds." In: *SECRYPT*. 2014, pp. 5–18.

[13] Amit Sahai and Brent Waters. "Fuzzy identity-based encryption". In: *Advances in Cryptology–EUROCRYPT 2005*. Springer, 2005, pp. 457–473.

[14] Vipul Goyal et al. "Attribute-based encryption for fine-grained access control of encrypted data". In: *Proceedings of the 13th ACM conference on Computer and communications security*. ACM. 2006, pp. 89–98.

[15] John Bethencourt, Amit Sahai, and Brent Waters. “Ciphertext-policy attribute-based encryption”. In: *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE. 2007, pp. 321–334.

[16] Jeremy Horwitz and Ben Lynn. “Toward hierarchical identity-based encryption”. In: *Advances in Cryptology—EUROCRYPT 2002*. Springer. 2002, pp. 466–481.

[17] Adi Shamir. “Identity-based cryptosystems and signature schemes”. In: *Advances in cryptology*. Springer. 1985, pp. 47–53.

[18] Mihaela Ion, Giovanni Russello, and Bruno Crispo. “Enforcing multi-user access policies to encrypted cloud databases”. In: *Policies for Distributed Systems and Networks (POLICY), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 175–177.

[19] Ben Adida. “Helios: Web-based Open-Audit Voting.” In: *USENIX Security Symposium*. Vol. 17. 2008, pp. 335–348.

[20] Luca Ferretti, Michele Colajanni, and Mirco Marchetti. “Distributed, concurrent, and independent access to encrypted cloud databases”. In: *Parallel and Distributed Systems, IEEE Transactions on* 25.2 (2014), pp. 437–446.

[21] Myrto Arapinis, Sergiu Bursuc, and Mark Ryan. “Privacy supporting cloud computing: Confichair, a case study”. In: *Principles of Security and Trust*. Springer, 2012, pp. 89–108.

[22] J. Callas et al. *OpenPGP Message Format*. RFC 4880 (Proposed Standard). Updated by RFC 5581. Internet Engineering Task Force, Nov. 2007. URL: `http://www.ietf.org/rfc/rfc4880.txt`.

[23] Marc Stevens, Pierre Karpman, and Thomas Peyrin. “Freestart collision for full SHA-1”. In: *Cryptology ePrint Archive* 2015/967 (2015), pp. 1–21.

[24] Bundesamt für Sicherheit in der Informationstechnik. “Technische Richtlinie TR-02102-1, Kryptographische Verfahren: Empfehlungen und Schlüssellängen”. In: *Report, Bundesamt für Sicherheit in der Informationstechnik (BSI)* (2016).

[25] Erik Nellessen. *Encrypting cloud storages*. `https://github.com/eriknellessen/encrypting-cloud-storages`. 2016.

[26] Ronald Rivest. “The MD5 message-digest algorithm”. In: (1992).

[27] PUB FIPS. “180-4”. In: *Federal Information Processing Standards Publication, Secure Hash* (2012).

[28] Taesoo Kim and Nickolai Zeldovich. “Practical and Effective Sandboxing for Non-root Users.” In: *USENIX Annual Technical Conference*. 2013, pp. 139–144.

[29] Mendel Rosenblum and John K Ousterhout. “The design and implementation of a log-structured file system”. In: *ACM Transactions on Computer Systems (TOCS)* 10.1 (1992), pp. 26–52.

[30] Dave Hitz, James Lau, and Michael A Malcolm. “File System Design for an NFS File Server Appliance.” In: *USENIX winter*. Vol. 94. 1994.

[31] Zakir Durumeric et al. “The matter of heartbleed”. In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM. 2014, pp. 475–488.

[32] Federal Office for Information Security (BSI). “Technical Guideline TR-03127: Architecture electronic Identity Card and electronic Resident Permit”. In: *Report, Federal Office for Information Security (BSI)* 1.13 (2011).

[33] Bundesamt für Sicherheit in der Informationstechnik. “Technische Richtlinie TR-03127: Architektur elektronischer Personalausweis”. In: *Report, Bundesamt für Sicherheit in der Informationstechnik (BSI)* 1.16 (2015).

[34] Federal Office for Information Security (BSI). “Technical Guideline BSI TR-03119 Requirements for Smart Card Readers Supporting eID and eSign Based on Extended Access Control”. In: *Report, Federal Office for Information Security (BSI)* 1.3 (2013).

[35] German Banking Industry Committee. “Secoder Application chipTAN”. In: *Specifications of the Secoder 3G* 1.1 (2015).

[36] Michael Roland. “Software card emulation in NFC-enabled mobile phones: Great advantage or security nightmare”. In: *Fourth International Workshop on Security and Privacy in Spontaneous Interaction and Mobile Phone Use*. 2012, pp. 1–6.

[37] Erik Nellessen. “Host-based Card Emulation of a PKCS15 compatible smartcard”. In: *SAR-PR-2014-08* (2014). `https://sar.informatik.hu-berlin.de/research/publications/SAR-PR-2014-08/SAR-PR-2014-08_.pdf`.

[38] Erik Nellessen. *User-controlled decryption operations*. `https://github.com/eriknellessen/user-controlled-decryption-operations`. 2016.

[39] Achim Pietig. *Functional specification of the OpenPGP application on ISO smart card operating systems*. 2004.

[40] RSA Laboratories. *PKCS #11: Cryptographic Token Interface Standard*. 2009.

Acronyms

ABE Attribute-Based Encryption. 16

APDU Application Protocol Data Unit. 53, 70

API Application Programming Interface. 17, 29, 44

CA Certificate Authority. 12, 72

CK Clearance Key. 24

CP-ABE Ciphertext-Policy Attribute-Based Encryption. 16

eID elektronischer Identitätsnachweis. 52–54, 70

EncFS Encrypted Filesystem. 39, 43–45, 48, 70

FUSE Filesystem in Userspace. 30, 43, 45, 48

GnuPG GNU Privacy Guard. 22, 37, 41–44, 48, 68, 69, 72

GPGME GnuPG Made Easy. 44, 45, 48, 68, 70, 71

GUI Graphical User Interface. 38

HCE Host-based Card Emulation. 68–70

HIBE Hierarchical Identity-Based Encryption. 16, 17

IBE Identity-Based Encryption. 16, 17

IP Internet Protocol. 12, 31

K2C Key to Cloud [10]. 16, 17, 20

KP-ABE Key-Policy Attribute-Based Encryption. 16, 17

K_B^p Bob's public key. 9, 10, 55

K_M^p The attacker's public key. 55

K_P^p The provider's public key. 9, 10

NFC Near Field Communication. 58, 59, 64, 66, 71, 73

nPA neuer Personalausweis. 52–54, 70

OpenPGP Open Pretty Good Privacy. 41, 42, 59, 68–71

PIN Personal Identification Number. 50, 53, 56, 58, 59, 61, 63–68, 70, 73

PKCS Public Key Cryptography Standard. 68

PKG Private Key Generator. 16, 17

QR Quick Response. 8, 11, 21, 23–25, 27–29, 40–42, 47, 56, 57, 72

TAN Transaction Authentication Number. 52–54

TCP Transmission Control Protocol. 12, 31

TLS Transport Layer Security. 21

TPM Trusted Platform Module. 58

U2F Universal Second Factor. 56, 59, 60

USB Universal Serial Bus. 58, 59

Appendix

Readme of the implementation of the first part

Welcome

This software enables you to transparently encrypt your Dropbox folder on your PC. It also supports sharing files with other Dropbox users, while still encrypting the shared data.

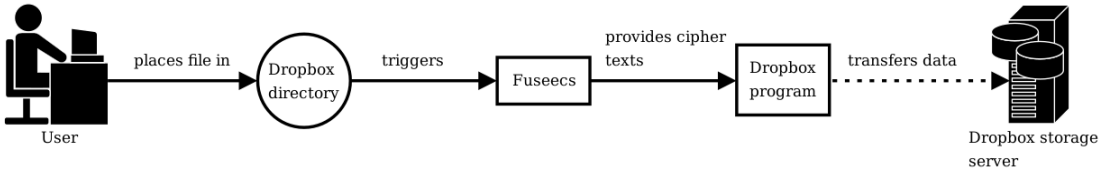

Figure 51: Overview of the involved components.

Warning: This is just proof-of-concept code and should *NOT* be used in production environments

Tested platforms:

- Debian Jessie (32 Bit)

Building

To build the software, execute the following commands:

```
git clone https://github.com/eriknellessen/encrypting-cloud-storages
cd encrypting-cloud-storages/build
cmake ..
make install
```

Using

Transparent client-side encryption

Setting up Dropbox

This needs to be done only once. It must be done before starting the transparent client-side encryption or Dropbox.

1. Create user Dropbox: `adduser Dropbox`

2. Install Dropbox (download *.deb from `https://www.dropbox.com/`)
3. Start Dropbox as normal user, so the files are installed. When it asks for your e-mail, close dropbox.
4. Grant user Dropbox write access to your home directory, e.g. by executing `chmod 777 ~`
5. Execute `xhost +` (as normal user)
6. Start Dropbox (as user Dropbox)
7. Choose your home directory when asked where to place the Dropbox directory
8. Terminate Dropbox
9. Reclaim your Dropbox directory via chown
10. Remove all files in Dropbox, e.g. by executing `rm -rf ./* ./.*` inside the Dropbox directory

Starting the transparent client-side encryption

This needs to be done before starting Dropbox.

To start the transparent client-side encryption, execute the following command:

```
bin/start_fuseecs.sh
```

Starting Dropbox

This must not be done before starting the transparent client-side encryption.

We need to share our display, so the user Dropbox can use it. We then switch to the user Dropbox and start the program:

```
xhost +
su Dropbox
/home/user/.dropbox-dist/dropbox-lnx.$PLATFORM-$VERSION/dropbox
```

Sharing files

For sharing a folder, execute the following command:

```
bin/start_share_a_folder.sh $FOLDER $OPENPGP_FINGERPRINT
```

For example, the command could look like this:

```
bin/start_share_a_folder.sh /home/user/Dropbox/folder_to_share \
A6506F46
```

This shares the folder in a cryptographic way. Afterwards, you still have to share the folder via Dropbox.

Readme of the implementation of the second part

Welcome

This software enables you to transparently encrypt your Dropbox folder on your PC. It also lets you confirm the decryption operations on an Android smartphone, which is used as an NFC-enabled token. Additionally, it supports sharing files with other Dropbox users, while still encrypting the shared data.

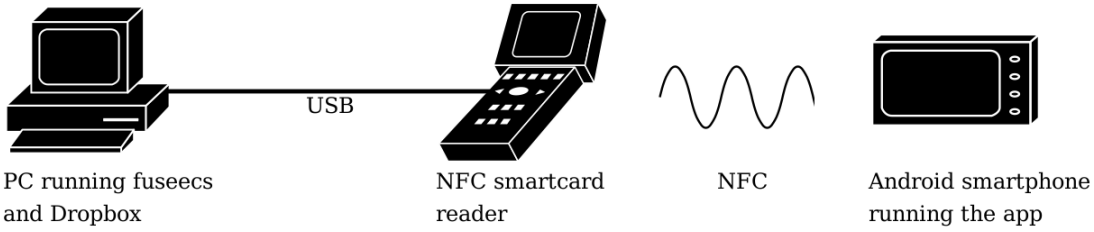

Figure 52: Overview of the involved components.

Warning: This is just proof-of-concept code and should *NOT* be used in production environments

Tested platforms:

- Debian Jessie (32 Bit) and Android Lollipop (5.0)

Building

To build the software, execute the following commands:

```
git clone https://github.com/eriknellessen/user-\
controlled-decryption-operations
cd user-controlled-decryption-operations
make
```

Using

Android App

Installing

To install the Android App on your smartphone, connect it to your PC, enable debugging and execute the following command:

```
cd Android
make install
```

Setup

We use the Android smartphone just like an NFC-enabled smartcard. So just place your smartphone on your NFC reader.

You now need to generate a key on the smartphone/push an existing key to the smartphone. Please notice, that the key is not saved to the next usage of the App, see `https://github.com/licel/jcardsim/issues/77`.

You can use `gpg` to generate/import the key. For a tutorial on importing keys to the smartphone, see `https://developers.yubico.com/PGP/Importing_keys.html`. For a tutorial on generating a key on the smartphone, see `https://www.gnupg.org/howtos/card-howto/en/ch03s03.html`.

Transparent client-side encryption

Configuring OpenSC

We need to configure OpenSC, so it chooses the right driver to communicate with our Android smartphone. To do so, add the following lines to the file `/etc/opensc/opensc.conf`:

```
card_drivers = openpgp-modified, internal;

card_driver openpgp-modified {
    # The location of the driver library
    module = /path/to/the/build/directory/lib/card-openpgp-modified.so;
}

card_atr 3b:80:80:01:01 {
    driver = "openpgp-modified";
}
```

On a 64 Bit system, you might need to change the path from

```
/path/to/the/build/directory/lib/card-openpgp-modified.so
```

to

```
/path/to/the/build/directory/lib64/card-openpgp-modified.so
```

Setting up Dropbox

This needs to be done only once. It must be done before starting the transparent client-side encryption or Dropbox.

1. Create user Dropbox: `adduser Dropbox`
2. Install Dropbox (download *.deb from `https://www.dropbox.com/`)

3. Start Dropbox as normal user, so the files are installed. When it asks for your e-mail, close dropbox.
4. Grant user Dropbox write access to your home directory, e.g. by executing `chmod 777 ~`
5. Execute `xhost +` (as normal user)
6. Start Dropbox (as user Dropbox)
7. Choose your home directory when asked where to place the Dropbox directory
8. Terminate Dropbox
9. Reclaim your Dropbox directory via chown
10. Remove all files in Dropbox, e.g. by executing `rm -rf ./* ./.*` inside the Dropbox directory

Starting the transparent client-side encryption

This needs to be done before starting Dropbox.

To start the transparent client-side encryption, execute the following command:

```
cd encrypting-cloud-storages/build
bin/start_fuseecs.sh
```

Starting Dropbox

This must not be done before starting the transparent client-side encryption.

We need to share our display, so the user Dropbox can use it. We then switch to the user Dropbox and start the program:

```
xhost +
su Dropbox
/home/user/.dropbox-dist/dropbox-lnx.$PLATFORM-$VERSION/dropbox
```

Sharing files

For sharing a folder, execute the following commands:

```
cd encrypting-cloud-storages/build
bin/start_share_a_folder.sh $FOLDER $OPENPGP_FINGERPRINT
```

For example, the commands could look like this:

```
cd encrypting-cloud-storages/build
bin/start_share_a_folder.sh /home/user/Dropbox/folder_to_share \
A6506F46
```

This shares the folder in a cryptographic way. Afterwards, you still have to share the folder via Dropbox.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 21. Juni 2016 .....