Technische Universität Berlin

Bachelor Thesis

Key Management and Key Exchange for Client-side Encrypted Cloud Storage

Valentin Franck, 364066

Supervised by
Prof. Dr. Stefan Tai
Prof. Dr. Sahin Albayrak

Advised by
Max-Robert Ulbricht

September 11, 2019

Eidesstattliche Versicherung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 11. September 2019.

.....
Valentin Franck

Abstract

Cloud storage systems have become very popular, rising questions about the security of the data shared in the cloud. This thesis presents a secure cloud storage system, that uses client-side encryption to protect the confidentiality, integrity and authenticity of the data in the cloud. It proposes a cryptographic tree structure that enables efficient and secure sharing of the data by using hybrid encryption. The cloud storage provider runs a Public Key Infrastructure, so that the user can easily retrieve other users' public keys to share data with them. To avoid putting too much trust in the cloud storage provider two certificate verification mechanisms are introduced: QR-Code verification and verification via the Socialist Millionaires' Protocol, which can be performed over a potentially insecure channel. In order to add further devices to one's account or recover from a device loss the user's private key is stored in the cloud encrypted with a random generated 12 English words passphrase. The feasibility of the presented design is evidenced by a proof of concept implementation and an evaluation.

Zusammenfassung

Cloudspeichersysteme sind sehr populär geworden, was Fragen nach der Sicherheit der Daten in der Cloud aufgeworfen hat. Diese Abschlussarbeit präsentiert ein Cloudspeichersystem, das Client-seitige Verschlüsselung zum Schutz der Vertraulichkeit, Integrität und Authentizität der Daten in der Cloud einsetzt. Die Arbeit schlägt eine kryptographische Baumstruktur vor, die durch den Einsatz hybrider Verschlüsselung, effizientes und sicheres Teilen der Daten in der Cloud ermöglicht. Der Cloudspeicheranbieter betreibt eine Public Key Infrastructure, so dass die öffentlichen Schlüssel anderer Nutzer_innen jederzeit abgerufen werden können, um Daten in der Cloud zu teilen. Um das Vertrauen in den Cloudspeicheranbieter zu minimieren, werden zwei Mechanismen zur Zertifikatverifizierung eingeführt: QR-Code Verifizierung und Verifizierung mittels des Socialist Millionaires' Protocol, das über einen potentiell unsicheren Kanal abgewickelt werden kann. Um ihrem Account weitere Geräte hinzuzufügen bzw. wieder Zugriff auf den Account nach dem Verlust eines Gerätes zu erhalten, werden die privaten Schlüssel von Nutzer_innen mit einer Passphrase aus 12 zufallsgenerierten englischen Wörtern verschlüsselt und in der Cloud gespeichert. Die Umsetzbarkeit des vorgestellten Designs wird durch eine Proof of Concept Implementierung mit Evaluation sichergestellt.

Contents

List of Figures

1 Introduction

In recent years cloud computing has become a huge hype in computer science. Both companies and private users have started to make use of the cloud in various ways– be it to outsource computing power, hardware maintenance or storage. One famous application provided by cloud computing technologies is reliable data storage.[1] In cloud storage systems users can store their files online, access them from any location and device while at the same time being able to share their data with others. This set of functionality is also known as Sync and Share. In spite of the convenience of cloud storages for many use cases (such as collaborative work on a larger set of documents, sharing or syncing (large) files or directories or a simple data backup) questions about privacy and security of the data processed and stored by the cloud storage provider (CSP) have been rising and preventing some users from actually using these services to their full potential.

Famous commercial CSPs, e.g. Dropbox and Google Drive, claim to be secure. Security for these CSPs mainly refers to their use of server-side encryption of the data at rest at the host and the use of transport encryption (SSL/TLS) between client and server and of course between different servers. While these security measures can provide some protection to keep data confidential, eventually it is based on the user's trust in the CSP itself. Since with server-side encryption the CSP (or its employees) may always access both encrypted data and the keys required to decrypt it, the CSP is free to read or even modify stored data at all times. The CSP may also pass on these rights without the user ever acknowledging it. Governments may legally force the CSP to provide customer data and of course in case of a server breach the users' data might also be compromised, since the encryption keys are right there next to the data.

Edward Snowden's revelations have brought some clarity to the actual extent to which the US foreign security agency NSA (among other organizations) applies techniques of mass surveillance to online communication. Snowden has made it very clear that the extensive use of state-of-the-art cryptography is the only way to protect one's privacy online. Since it is not only governments but also private companies (with the intention to analyze the users' data and behavior to improve advertisement) and for example criminal hackers (deploying ransom ware etc.) that might seek access to the data stored in the cloud, server-side encryption is not enough to be counted as an appropriate security measure. Groups of activists, lawyers and journalists, that heavily rely on the confidentiality of their communication are particular targets for state surveillance, which may in some occasions put them at great risk when using insecure cloud storages. If they are aware of the threat, this deprives them from using convenient communication tools and thus is a major hindrance to their work. So particularly targeted groups are in real need of tools that allow them to sync and share their data and resolve the mentioned shortcomings regarding security of the most popular CSPs. [2]

[1] E.g. even the Amazon S3 Reduced Redundancy Storage stores data various times redundantly on different machines and thus achieves a much higher availability than traditional backup solutions (Amazon 2018).

[2] Mobile Messengers such as WhatsApp have reacted to that threat by applying the Signal Protocol, a modern cryptographic protocol for asynchronous messaging, that provides "end-to-end

Current client-side encrypted[3] cloud storages suffer from several issues. In some cases encryption is realized using containers instead of using different keys for each file. That means, the whole container has to be re-encrypted and uploaded again, every time the user just modifies one single file. More importantly there are severe security problems with existing client-side encrypted cloud storages. Some solutions do not allow users to share data at all, while others that do, require users to trust the CSP, that data is really only just shared with the person the CSP claims. The latter problem arises, because these CSPs use a Public Key Infrastructure (PKI), in which they act both as Certificate Authorities (CA) and key distributors. In this scenario, if Alice wants to share a file with Bob, she has to trust the CSP, that she really received Bob's public key and that the CSP is not performing a man in the middle attack.[4] But there is another problem in existing solutions. Either users are not able to use multiple devices, because they cannot transfer their private keys or the server is responsible for the transfer. Since these private keys are at best encrypted with the password of the user account (which is usually relatively weak), the server might even get direct access to the user's private key and thus to all data she stores in the cloud.

This thesis presents solutions to all three problems. To organize encryption efficiently we use a cryptographic tree structure as first proposed by Grolimund et al. (2006), which resembles the folder structure. In this tree structure every file and every folder is encrypted with a new key. Every file and folder key is encrypted with the parent directories folder key up to the user's root directory. The key for the root directory is encrypted with the user's private key. In order to share a folder with Bob Alice simply has to decrypt the folder key and re-encrypt it with Bob's public key. Bob can then use the folder key to recursively decrypt all contained file and subfolder keys. This protects the confidentiality of the data. A second cryptographic tree structure is introduced to distinguish between read and write access. In this second write tree a private key is stored for every file and every folder. Whenever Alice or Bob modify a file, they use this key to sign the new version, so that anybody with read access can verify the signature, because the corresponding public key is stored in the read tree. By using digital signatures we protect the authenticity and integrity of the shared data.

---

security, deniability, forward secrecy, and future secrecy," (Frosch et al. 2016) while not making concessions to user experience.

[3]In this work we will use the terms client-side and end-to-end encryption mostly synonymously in the context of cloud storage. By both we refer to a system that encrypts the users' data on the users' devices before uploading it to the server. The server does not get access to the keys in plaintext and therefore cannot access the data either. Sometimes client-side encryption is referred to as a system that applies both end-to-end encryption and local encryption of the data at rest at the client. The term end-to-end encryption originally comes from software more focused on the exchange of messages such as email and instant messaging. Since there are plenty of solutions for local encryption, that can easily be implemented on top of our system, we will not go into detail about local encryption.

[4]In a man in the middle attack the channel between the communicating parties is controlled by an attacker. The attacker may have the ability to read, modify, delete and inject messages. The detection of such attacks is not easy and their mitigation usually requires a shared secret or an out of band channel between the communicating parties. This is why it is so important to verify the counterpart's public keys.

The solution we present in this thesis does not require any trust in the CSP. The CSP provides a PKI, in which it is the CA and we also make leverage of the availability of the CSP for key distribution. So whenever Alice wants to share a file with Bob she can retrieve Bob's certificate issued by the CSP. Alice is now technically able to share data with Bob. To make sure Alice really has Bob's valid certificate and not one controlled by the CSP, we provide two possible verification mechanisms: QR-Code verification and verification via the Socialist Millionaires' Protocol (SMP). For QR-Code verification Alice and Bob have to meet or Bob has to send Alice the QR-Code generated on his device. Alice can scan the QR-Code and if the scan succeeded, Alice and Bob have mutually verified that they are really using each other's certificates. The SMP on the other hand is a cryptographic protocol that allows Alice and Bob to mutually verify their certificates using a potentially insecure in-band channel. They only need to share a weak natural language secret, like the name of the restaurant, they had dinner at on their last vacation.
In order to allow users to recover from private key loss and give new devices access to their account, users have to exchange the private key between devices. For this purpose we randomly select 12 words from a 2048 English words list. We then apply a key derivation function and use the resulting key to symmetrically encrypt the private key. In this encrypted form the private key is stored in the cloud. So if Alice wants to add a new device she just downloads her own private key from the cloud storage and uses the passphrase only she knows to decrypt it. She then has gained full access to her account. Using this 12 word passphrase enforces passwords with enough entropy while not requiring complicated interaction from the user.
With our application scenario in mind it is important that we achieve our security goals without sacrificing usability, i.e. the Sync and Share functionality should be preserved. Our proof of concept implementation shows that we achieved this goal with only one notable, but deliberate exception. Web access is not supported by the design we chose. Since there are currently no easy ways to provide web access in a truly secure manner, we decided not to include it, but it could easily be added for convenience sacrificing some of the achieved degree of security, as we will explain in section 5. Apart from usability we consider it a crucial property of our software to be open-source (or at least have frequent external security audits), because otherwise users will never be able to confirm that the encryption actually works as stated by the CSP and that there are no back doors, which might be used to access the confidential data.[5]
Throughout this thesis we will elaborate our system, beginning in section 2 by describing the requirements for a cloud storage system and especially our threat model. In section 3 we will explain the cryptographic basics, because the security of our design heavily depends on using secure algorithms and implementing them correctly in software. Section 4 analyzes the related work and also provides an overview over other client-side encrypted cloud storages analyzing their strengths and weaknesses. In section 5 we explain why web access and sharing via encrypted web links would be a security threat and therefore is not supported by the proposed

[5]Note that the open-source cloud storage software Nextcloud has introduced end-to-end encryption during the elaboration of this thesis. It is a good approach, but unfortunately does not provide optimal usability and lacks proper key verification mechanisms (see section 4.2.6).

system. In the next three sections we present the design choices for our own design as outlined above. Section 6 explains the cryptographic tree structures that enable flexible sharing, modification of files and revocation of shares. The details of how the cross-device key exchange works are described in section 7. In section 8 we argue why verification mechanism are important and present the ones we decided are best suitable for client-side encrypted cloud storages. Finally in section 9 we present the details of our proof of concept implementation and discuss our results in section 10.

2 Requirements for a secure cloud storage

In this section we will briefly explain the application scenario and use cases the cloud storage system proposed by this thesis is designed for. We will then describe our threat model and clarify the essential security goals of our design. We will finish this section with a short explanation, why usability is a crucial property for our design (and should be for any secure privacy preserving software).

2.1 Application scenario and desired functionality

The application scenario that we have in mind is to provide secure cloud storage to end users and small-scale organizations, such as Non-Governmental-Organizations (NGOs), small groups of activists, journalists, lawyers and other associations, that are particular targets of (state) surveillance and repression. This means the focus of the design is rather on privacy and security than performance and scalability, as long as the system remains usable.
Typical use cases for the system are as follows:

- Data storage and backup: the user uploads personal data, in order to keep a backup or to be able to access the data online. This can be anything from single documents to a complete hard drive image.
- Syncing data: the user can use the cloud storage system to automatically sync data between several of her devices.
- Sharing data: the user uploads data, in order to share it with other users. This may be single files or complete hierarchical directories. The user wants it to be as easy as possible to decide which other users get access to the data. The user determines, what kind of access rights other users are granted. The user also wants to be able to revoke these access rights any time.

These are just the most common use cases as intended by our design. This list does not pretend to be exhaustive. Nevertheless it is enough to give us an idea about how cloud storages are used and what the consequences of a compromise of confidential data might be.
The cloud storage might be run on a private or on a public cloud, e.g. small groups of activists might even use a shared host to run the cloud storage server, while others might rely on public cloud services such as Amazon S3, because it provides elastic and reliable storage. Note that the server-side of our system is intended to work as

a middleware on top of an existing cloud storage, which is why basic cloud storage functionality is merely an abstraction to our design.

2.2 Basic security goals

There are several core concepts in information security referring to different protection goals a secure system should comply with. The best known is the "CIA triad", confidentiality, integrity and availability. For our context we will add authenticity as a key property.

- *Confidentiality*: No one is able to learn any information about the user's data without her authorization.
- *Integrity*: Any unauthorized modification of the user's data (including creation and deletion of data) will be detected (and at best can be rolled back).
- *Availability*: The user's data is (*efficiently*) accessible at all times from any machine by an authorized user.
- *Authenticity*: Any communicating entity really is, who it claims to be and the author of any modification of the system can be identified.

In this thesis we will not answer the question, how to protect the availability of the users' data due to our threat model, in which we assume that the server will not attack availability of the users' data (see the following section).
Because it is crucial "that the security properties described above are achieved based on strong cryptographic guarantees as opposed to legal, physical and access control mechanisms" (Kamara and Lauter 2010, 137), in section 3 we will take a closer look at the required cryptographic basics. Kamara and Lauter (2010, 138) also point out that "it is important that the client-side application and, in particular, the core components be either open-source or implemented or verified by someone other than the cloud service provider." This is why we also make it an essential requirement for the client application to be open-source.

2.3 Threat model

In this section we will develop a threat model. Credit is given to Mathieu Bourrier (2015), on whose threat model for a proposed client-side encrypted ownCloud Desktop Client, we build our own threat model. First of all we define that any of the user's or a friend's devices (and the client it runs) is trusted. The user needs to trust her own devices and also those of her friends in order to share data with them. If the user's device was compromised, it could easily, send all plaintext to an attacker (cf. Wilson and Ateniese 2014, 406). That means, if the user's account is compromised or data is compromised on a trusted device, due to local vulnerabilities, this is not considered a breach of our security model.[6] However, a user's account must not be

[6]Any user concerned about the security of her data should take further security measures, such as local file encryption, system hardening etc. The current debate on source surveillance

compromised due to a weakness in the client-side encrypted cloud storage system itself.
All networks, the data passes on its way between cloud storage and client are considered potentially insecure and attackers might actively try to listen in or even modify data on those networks. This problem can easily be resolved by enforcing transport encryption (TLS) (cf. Bourrier 2015, 5) and is therefore not contemplated in detail in this thesis.
As a typical scenario we assume that an attacker wants to get read access to the users' data stored on the cloud storage system. The attacker, the proposed design pretends to protect against, may be a government institution trying to read the documents stored and shared by activists, journalists and lawyers on the cloud storage system. This means the attacker has many (legal and financial) resources. We assume that the attacker will mostly try to remain covert, as governments usually do when monitoring their surveillance targets. Note that this is not necessarily the case and in some situations attackers might also try to manipulate the stored data or in case of subpoenas legally force the CSP to collaborate to provide access to user data. The consequence of these consideration is, that the CSP cannot be trusted to protect and respect the confidentiality of the users' data. However, we assume that it will stick to the protocol and most importantly guarantee the availability of the users' data.[7] Furthermore for the case of a server breach, an attacker with access to the host might try to modify user data, even if the attacker cannot access the plaintext. This means a government will mostly act as an honest but curious attacker, that might have access to the CSP.
While it has become a standard not to trust the network, not trusting the host requires security measures far more elaborated than transport encryption. Even if an attacker controls the host, she should not be able to access or modify any user's plaintext data without knowledge of the user. Yet, we consider it acceptable, if such an attacker is allowed to be able to get some limited information on the metadata, e.g. an estimation of the size of the data, the number of files and which users access the data.[8] An attacker controlling the host might also be able to provide outdated versions of files to users. This might even include providing different versions to different users (Cachin, Keidar, and Shraer 2009). However, this is an unlikely scenario with relatively low risks for the assets since it requires a lot of effort from the attacker, who would not even be able to know the difference between these versions, if the confidentiality of the data is adequately protected. Our design does

in telecommunication has pointed out, that given the increasing use of transport encryption and end-to-end encryption, governments are willing to gain direct access to the telecommunication data on the source devices. Therefore governments are currently working on the legal framework, that would give them a lot of competences to infiltrate devices for surveillance (Meister, Andre 2017).

[7]Availability and reliability are some of the main reasons users migrate their data to cloud storages. It is therefore highly unlikely that a CSP does not protect availability. If we wanted to enforce the availability of the data, we would need a completely different design. This would probably mean, that we could not trust a single CSP, but had to redundantly store our data at different cloud storages, at best working under different legislations. This is not the subject of this thesis.

[8]Some of these information leaks, such as analyzing cooperation networks of users might possibly be prevented by the application of modern functional cryptography. This is not the focus of our scenario. For further reading see: Kamara and Lauter 2010.

not protect against such an unlikely attack.
One crucial asset of a cloud storage is the users' data, which under all circumstances must remain confidential. Unauthorized (read) access to the stored data is considered to have the most severe consequences and therefore represents the greatest risk according to our application scenario. Therefore our system aims to protect the integrity, confidentiality and authenticity of the data stored in the cloud and in transmission. It does not protect the availability in the cloud and does not protect the data stored on the users' devices.

2.4 Usability

In their case study Whitten and Tygar (1998) evaluate the usability of security software and come to a clear conclusion: The more efforts a technology requires users to make, the less likely they are going to adopt it. In software engineering usability "is a quality attribute that assesses how easy user interfaces are to use. The word 'usability' also refers to methods for improving ease-of-use during the design process. Usability is defined by 5 quality components" (Nielsen 2017):

- "*Learnability*: How easy is it for users to accomplish basic tasks the first time they encounter the design?
- *Efficiency*: Once users have learned the design, how quickly can they perform tasks?
- *Memorability*: When users return to the design after a period of not using it, how easily can they reestablish proficiency?
- *Errors*: How many errors do users make, how severe are these errors, and how easily can they recover from the errors?
- *Satisfaction*: How pleasant is it to use the design" (Nielsen 2017).

While utility describes, whether a software provides the needed functionality, usability describes how easy it is to actually benefit from these features (Nielsen 2017). The utility of a client-side encrypted cloud storage has been sufficiently described. Therefore in order to make it useful we need to make sure that the proposed system provides good usability. In a market with plenty of CSPs available, solutions that do not provide good usability will be discarded. To achieve our goals and promote the general use of privacy preserving cloud storages usability is a key factor.[9]
In this thesis we do not want to go into the details of the research that has been done in the field of usability, but rather take a practical approach to it. We assume, that the existing cloud storage solutions (to a slightly varying degree) already provide good user experience. So our main goal will be to preserve the usability and utility (namely the Sync and Share functionality) of these systems. We will therefore make

[9]Usability, is sometimes used interchangeably with the term user experience, which denotes the emotional and more subjective experience a user makes while interacting with the software. Usability is usually considered a prerequisite for a good user experience, because no user will ever make a positive experience, if she cannot get her task done (efficiently) (Law et al. 2009).

the encryption fully transparent where possible and make usability a priority in any design decision. However in certain circumstances full transparency will not be possible without sacrificing the gain in security our design aims at. In these cases we will limit the additional user interaction to a minimum and make it as easy to understand as possible. By transparency we mean, that the user will not be able to tell whether the files are actually encrypted, because no interaction from her part is required and she can use the cloud storage, just as if there was no encryption at all. For the most part the system we propose in this thesis will therefore not sacrifice user experience. However, some security measures such as certificate verification require user interaction and in the case of web access we even decided not to include it at all.

3 Cryptographic basics and security infrastructures

In this section we will discuss the cryptographic basics required to design a secure cloud storage system. We will sketch out how the discussed protocols work, what their pitfalls are and whether they are currently considered secure. The focus will be on those protocols and algorithms, that will later be suggested for use in a client-side encrypted cloud storage system.

3.1 Symmetric-key cryptography

Symmetric-key cryptography is the oldest form of cryptography. It is still widely used because of its relative simplicity and far better performance than asymmetric cryptography.

> "Among all encryption algorithms, symmetric-key encryption algorithms have the fastest implementations in hardware and software. Therefore, they are very well-suited to the encryption of large amounts of data" (Delfs and Knebl 2015, 11f).

Symmetric encryption is also used in hybrid cryptographic systems, in order to benefit from both the performance of symmetric-key cryptography and the easement of the key negotiation or exchange enabled by public key cryptography. Modern symmetric-key cryptography is not only used for encryption and decryption, but also in cryptographic hash functions. In symmetric-key encryption the communicating parties, say Alice and Bob, first have to agree on a shared key, which they keep secret. Alice uses that key to encrypt a message to Bob, who uses the same key to decrypt it. Although the algorithms for encryption and decryption are publicly known, nobody should be able to extract any information about the plaintext of a message from its ciphertext without knowledge of the encryption key. Because both parties use the same key for encryption and decryption, we speak of symmetric-key encryption. The main challenge in this scheme, i.e. how to exchange the shared key, has led to the invention of public key cryptography. There are basically two different types of symmetric encryption algorithms:

> "Block ciphers operate on data with a fixed transformation on large blocks of plaintext data; stream ciphers operate with a time-varying transformation on individual plaintext digits" (Rueppel 1992).

In this paper we will only explain a block cipher algorithm, AES, since block ciphers are most widely used in software today, because they can be implemented more efficiently in software (Schneier 1996, 211).

3.1.1 Advanced Encryption Standard (AES)

AES is a relatively new encryption algorithm.[10] AES is an iterated block cipher, that has a fixed block length of 128 bits and a key length of 128, 192 or 256 bits. Depending on the key length the algorithm applies a round function 10, 12 or 14 times after one initial round key addition. The algorithm first creates a state, represented as a matrix and then round-wisely applies byte substitution, row shifting and column mixing. However the only operation depending on the secret key is the addition (XORing) of the initial key and of a round key in every round. Those keys are derived from the secret key by a key expansion function.
The algorithm is obviously invertible, because XORing can be inverted by simply applying it twice and the other operations can simply be reverted. The round function ensures some crucial properties of AES. The byte substitution causes nonlinearity (confusion between ciphertext and secret key) and the row shifting provides diffusion. i.e. every bit change in the key results in a 50% chance of a bit change in every bit of the ciphertext and every bit change in the plaintext results in a 50% chance of a bit change in every bit of the ciphertext (Ferguson 2010, 55).
Although the focus of AES is mainly on performance, it is reasonably secure to use. However, there have been certain advances towards breaking AES:

> "The attacks against the full 192- and 256-bit versions of AES are theoretical–not practical–so we aren't ready to lose any sleep over them just yet. [...] Given all that we know today, using AES still seems like reasonable decision. It is the U.S. government standard, which means a great deal. Using the standard avoids a number of discussions and problems. But it is important to realize it is still possible that future cryptanalytic advances may uncover even more serious weaknesses" (Ferguson 2010, 55f).

However, until today none of these first approaches to break AES has led to success in practice which is why is why Pancholi and Patel (2016) consider that the "AES algorithm is a highly secure encryption method", that has a "high performance without any weaknesses and limitations."
Due to its good performance and security we will use AES for symmetric encryption in the client-side encrypted cloud storage proposed by this thesis. Another major advantage of AES is that it is "the standard" and therefore supported by most cryptography libraries (Ferguson 2010, 59).

[10] Since its predecessor DES was not regarded to be entirely secure anymore the Nation Institute of Standards and Technology (NIST) started a selection process in 1997 for the new Advanced Encryption Standard (AES). Proposals could be made and finally three years later the Rijndael algorithm was chosen in a public process.

3.1.2 Hash functions

Cryptographic hash functions are one application of symmetric cryptography. A hash function is a one-way function, which means that it is easy to compute the hash value $H(M)$ of any given message M, while it is practically infeasible to compute the original message M just from knowledge of its hash value $H(M)$ and the hash function H. Given M it should also be hard to find another message M' with $M \neq M'$ and $H(M) = H(M')$. For some applications collision-resistance is also needed: it is practically infeasible to find two messages $M \neq M'$ with $H(M) = H(M')$ (Schneier 1996, 429). [11]

3.1.3 Keyed-Hash Message Authentication Code (HMAC)

Message Authentication Codes (MAC) are similar to cryptographic hash functions. In the case of MACs a hash function is used to compute a unique fingerprint (of fixed length) of a given message. This hash value is encrypted with a secret key. Since it is impossible to change the message and recompute a valid MAC without knowledge of the key, MACs ensure the integrity and authenticity of the message. A MAC can also only be verified by someone with access to the secret key (Schneier 1996, 455f).
The most commonly used method today to create a MAC from a hash function is the Keyed-hash MAC (HMAC). HMAC does not specify one particular hash function, but can be used with different hash functions such as MD5 or the SHA-family. To guarantee the security of the MAC, the hash function is applied twice.[12] The HMAC of a message M is computed as follows:
$\text{HMAC}(K, M) = H((K \oplus opad)||H((K \oplus ipad)||M))$ where K is the key, H is the used hash function and *ipad* and *opad* are the constant inner ($0x36$ repeatedly) and outer padding ($0x5C$ repeatedly) used to XOR the key with. *ipad* and *opad* are used to derive two different keys for the concatenation with the message and its hash value respectively in the inner and outer application of the hash function (Delfs and Knebl 2015, 42ff).
One of the advantages of HMAC over traditional MACs is the variability of the length of the resulting HMAC depending on the used hash function. It is easy for the receiver of the HMAC to verify, that the sender knows the shared secret key and the message has not been modified. On the other hand it not possible to find a message M for a given HMAC, due to the one-way property of cryptographic

[11] This is a stronger requirement than the one mentioned before and needed to avoid birthday attacks (Schneier 1996, 165f). Some cryptographic hash functions such as MD5 are no longer considered to be collision-resistant and should therefore not be used in digital signatures and other critical applications. In 2005 the Chinese scientists Wang, Yin and Yu found a way to ease the finding of collisions in SHA-1. However, the improved algorithms of SHA-2 (with 256, 384 and 512 bit hash value length) are still considered to be collision-resistant. Encouraged by the great success of AES the NIST started a similar selection process for a new secure cryptographic hash algorithm SHA-3 in 2007. In 2012 Keccak's algorithm was chosen. Because SHA-2 is still viewed as secure and is included in almost all cryptographic libraries, the use of SHA-2 is suggested in this thesis (Delfs and Knebl 2015, 40ff).

[12] This prevents the length extension attack, where an attacker with knowledge of $\text{MAC}(M)$ is also able to compute the MAC for another message M' attached to M: $\text{MAC}(M||M')$ (Swoboda, Spitz, and Pramateftakis 2008, 107).

hash functions. This is indeed possible for other MACs under certain circumstances (Swoboda, Spitz, and Pramateftakis 2008, 108). Due to these characteristics HMAC can be viewed as a secure way to ensure authenticity and integrity in messages provided that the communicating parties share a secret key.

3.2 Password-based key derivation functions

In many security applications it is not possible (or at least not desirable regarding usability) that the user always has to provide her secret key. While long secret keys are impossible to remember or to transfer to another device, passwords are not. Password-based key derivation functions have been developed to derive secret keys from a user's password. One use case of password-based key derivation functions is calculating a hash value from the password that is hard to attack using brute force, rainbow tables or dictionary attacks.[13] In this section we will briefly discuss, how PBKDF2 works and what degree of security it provides. The problem with passwords being the basis of a secret key is obvious:

> "[P]asswords, in particular those chosen by a user, often are short or have low entropy. Therefore special treatment is required in the key derivation process to defend against exhaustive key search attacks" (Yao and Yin 2005).

3.2.1 PBKDF2

Generally key derivation functions use the following scheme to derive the key K: $K = H^C(P||S)$. H is a function, that is applied C times to the concatenation of the password P and the salt S. In the industry standard PKCS#5 two different functions PBKDF1 and PBKDF2 are defined. H in PBKDF1 is a hash function such as MD5 or SHA-1 and the salt S should be at least 64 bits long, while the number of iterations C should at least be 1000. PBKDF2 is designed to be more secure and uses a keyed-hash function, such as HMAC, instead of a simple hash function. The password in this case is used as the key. Adding some extra protection by XORing the results of each iteration the key is derived as follows: $K = H_P^1(s) \oplus H_P^2(s) \oplus ... \oplus H_P^C(s)$ A high number of iterations makes sure that it is costly to compute the key from a password. By the concatenation with a salt the entropy of the password is further increased, which makes it practically infeasible to build rainbow tables. The application of both methods therefore significantly decreases the success chances of brute force and dictionary attacks. In their paper Yao and Yin (2005) show that the mechanisms used in PBKDF2 indeed make this function secure in case that the attacker cannot choose the parameters such as the salt and the number of iterations. Otherwise the function may have serious vulnerabilities. Colin Percival, the inventor of the password-based key derivation function Scrypt, points out that users tend to choose weak (low-entropy) passwords:

> "Unfortunately, this form of "brute force" attack is quite liable to succeed. Users often select passwords which have far less entropy than is typically

[13] These are the only three feasible attacks against modern password-based key derivation functions, because they are based on strong hash functions (Percival 2009, 1).

> required of cryptographic keys; a recent study found that even for web sites such as paypal.com, where — since accounts are often linked to credit cards and bank accounts — one would expect users to make an effort to use strong passwords, the average password has an estimated entropy of 42.02 bits, while only a very small fraction had more than 64 bits of entropy [15]." (Percival 2009, 1)

The use of password-based key derivation functions should therefore be considered carefully for each particular application scenario, because otherwise users are able to sacrifice the security of the complete system. This is why we will force user to encrypt their private keys with a random generated 12 words high entropy password (see section 7).

3.3 Public key cryptography

The invention of public key cryptography was one of the most important advances in cryptographic methods that was ever made. It was first introduced in the mid 1970's by Diffie and Hellman (1976), who published their still widely used key exchange algorithm and then by Rivest, Shamir, and Adleman (1978), who published the algorithm today best known by their initials: RSA. Prior to the invention of public key cryptography it was impossible to exchange or negotiate symmetric keys over an insecure channel. Also, each pair of communicating parties needed their own secret key, which meant that it did not scale for large communicating groups.[14]
The basic idea of public key cryptography is that each communicating entity owns two keys. One that is public and accessible for everyone. The other one that is private and must always remain the secret of this entity. If Alice wants to send a message to Bob, she uses Bob's public key to encrypt it and then sends it to him. Upon receiving Alice's message Bob uses his own private key to decrypt it. That means, that anyone can encrypt a message using Bob's public key, but only someone who possesses the corresponding private key, that is Bob, will be able to decrypt it. It is crucial, that knowledge of the ciphertext, the plaintext of the message and the public key used for encryption do not allow any conclusions about the private key, since this would compromise all other messages encrypted with this private/public key pair. It must also be infeasible to get any useful information on the plaintext of a message by mere knowledge of its ciphertext and the public encryption key. The first algorithm, that allowed such characteristics was RSA. Later, however, certain improvements were made by similar approaches such as elliptic-curve cryptography, which is based on a similar mathematical problem as RSA.[15]
Since public key cryptography is far slower and uses a lot longer key lengths than conventional symmetric-key cryptography, it was not immediately clear, how crypto systems could benefit from public key cryptography:

> "In viewing public key cryptography as a new form of crypto system rather than a new form of key management, we set the stage for criticism

[14] If there are n communicating parties, the number of keys to be exchanged is in $O(n^2)$.

[15] We will neither explain how elliptic-curve cryptography nor how a Diffie-Hellman Key Exchange work, because both are only used in the transport encryption our design uses.

> on grounds of both security and performance. Opponents were quick to point out that the RSA system ran about one-thousandth as fast as DES and required keys about ten times as large. Although it had been obvious from the beginning that the use of public key systems could be limited to exchanging keys for conventional cryptography, it was not immediately clear that this was necessary. In this context, the proposal to build *hybrid* systems was hailed as a discovery in its own right" (Diffie 1992).

This means that while symmetric-key cryptography is best suited to encrypt data due to its performance and security, public key cryptography should be used for key management and key exchange in particular, thus making a hybrid system the most secure and effective on we can think of today. In a hybrid system the data is encrypted with a random symmetric key, that in turn is encrypted with the receiver's public key. Today public key cryptography has its well-established field of application, that does exceed encryption, e.g. it is also used for digital signatures, public key certificates etc. (cf. Schneier 1996, 216, 461).

3.3.1 RSA

Of all public key algorithms RSA is probably the most popular. This is on the one hand due to its early publication and on the other hand due to the ease of understanding and implementation it provides (Schneier 1996, 466f). It is still used in a wide number of applications such as PGP or the Diffie-Hellman Key Exchange in TLS. However, nowadays elliptic curves have replaced RSA in many applications because of their efficiency and the relatively small length of the required public keys. The security of RSA is based on the difficulty to factorize large numbers. For key generation the user chooses two large prime numbers p and q of equal length (in bit representation) and then computes the product $n = pq$. The (public) encryption key is randomly chosen "such that e and $(p-1)(q-1)$ are relatively prime" (Schneier 1996, 467). The decryption key d is computed with the Euclidean Algorithm: $ed \equiv 1 mod\ (p-1)(q-1)$. This means that d and n are also relatively prime. The public key thus consists of the tuple (e, n) and the private key of the tuple (d, n).[16] For encryption a message m is divided into blocks of a fixed length, depending on the length of p and q. The ciphertext of the i-th block is computed as follows: $E(m_i) = c_i = m_i^e mod\, n$. Decryption is just the inversion of the encryption operation: $D(c_i) = m_i = c_i^d mod\ n$ (Schneier 1996, 466ff).[17]
RSA, if implemented correctly, is still a secure algorithm, since the best attack mathematically known is to factorize n. However it has not been proven that the security of RSA is based on the factorization of large numbers. So, theoretically, there might be an easier attack. The factorization of large numbers, however, is believed to be a hard problem, that has no polynomial solution. This has not been proved either (Ertel 2012, 82f). A chosen plaintext attack on public key crypto systems is easy. Given a ciphertext c encrypted with Alice's public key e_A Eve can

[16] In practice the Chinese remainder theorem is used to improve the performance and therefore even more numbers are included in the private key.

[17] For a proof that $D(E(m_i)) = E(D(m_i)) = m_i$ see Ertel 2012, 81.

encrypt plaintext messages m using e_A until she finds one with $E_{e_A}(m) = c$. This m is the original message. However this attack can easily be prevented by a good implementation that uses randomized padding before encrypting the message, so that the same plaintext is mapped to different ciphertexts. A chosen ciphertext attack combined with social engineering is also possible, but practically extremely unlikely to be successful, which is why RSA can still be considered a secure algorithm (Ertel 2012, 87).

3.3.2 Digital Signatures

Public key pairs, such as RSA keys, cannot only be used for encryption but also for digital signatures. If Alice wants to digitally sign a message to Bob, she computes the hash value of the message, encrypts the hash value with her private key and attaches it to the message. Upon receiving Alice's message Bob decrypts the hash value with Alice's public key. Bob computes the hash value of the message he received and compares it to the one Alice sent him. If they match, he has successfully verified the authenticity and integrity of the message he received. This works because of the equation $D(E(m_i)) = E(D(m_i)) = m_i$. Both, public and private key can be used for encryption and decryption, that means that encryption and decryption are inverse functions.[18]

3.4 Public key verification

The invention of public key cryptography has certainly been a crucial step towards today's wide use of cryptography in all kinds of applications. It has made key exchange a lot easier. However, it has brought its own kind of difficulties. The challenge for Alice, when sending a message to Bob, is how to get Bob's public key and how to verify that it really is Bob's key and has not been replaced by some attacker, who is pretending to be Bob.[19]
There are three common ways for Alice to get Bob's public key (cf. Schneier 1996, 185f):

- get it from Bob directly
- get it from a centralized database
- get it from her own database

Alice has different options to verify, that the public key she received really belongs to Bob. The first one is to meet with Bob in person (or call him) and verify that the fingerprint of the key matches. A fingerprint is a hash value, that is unique for every key. The important part about this method is, that an out-of-band channel is used, which is unlikely to be controlled by an attacker. The use of an out-of-band channel is not always possible or practical. Therefore Webs of Trust and Public-Key-Infrastructures have been developed, which we will take a look at in the following sections.

[18] Depending on the application scenario different key pairs should be used for signatures and encryption (Schneier 1996, 483ff).

[19] How a so called man in the middle attack works is described in section 8.1

3.4.1 Web of Trust

A Web of Trust works as follows. If Alice cannot make sure personally, that she has Bob's public key, she may still have enough confidence in having the right key, if someone she trusts, tells her it is the right key. In a Web of Trust users sign other users' public keys, once they have verified the keys. So if Alice has verified Bob's public key and Bob has signed Carol's key. Alice can decide that, she will trust Carol's key too, since she trusts Bob and has verified his key. When a lot of users are involved in this and sign each other's keys, this creates a Web of Trust.[20] There is one obvious problem with Webs of Trust: they require a lot of active contribution and a lot of awareness for security issues from the users. Even if Alice believes Bob would not maliciously sign Carol's wrong key, can she be sure, that Bob put in all the necessary work to verify it was really Carol's key, he signed? Alice certainly can never be sure and will be so even less, if she does not know Bob personally. The second problem is, that Bob in this scenario actively has to decide, whether he wants to sign Carol's key or not. After signing it he has to upload the signed key to a key distribution server or send it back to Carol. This makes is a rather cumbersome process (cf. Ertel 2012, 120f). Both these problems made Webs of Trust an acceptable solution, when they were first introduced at the invention of PGP in the 1990's, but there are more suitable solutions for our purposes.

3.4.2 Public key infrastructures (PKI)

Today the most common solution for the problem of key verification are Public Key Infrastructures (PKI). The purpose of a PKI is to allow us to tell, which entity a public key belongs to.

> "There is a central authority that is called the *Certificate Authority*, or CA for short. The CA has a public/private key pair (e.g., an RSA key pair) and publishes the public key. We will assume that everybody knows the CA's public key. [...] Alice generates her own public/private key pair. She keeps the private key secret, and takes the public key PK_A to the CA and says: 'Hi, I'm Alice and PK_A is my public key.' The CA verifies that Alice is who she says she is and then signs a digital statement that states something like 'Key PK_A belongs to Alice.' This signed statement is called the *certificate*. It certifies that the key belongs to Alice.
> If Alice now wants to communicate with Bob, she can send him her public key certificate. Bob has the CA's public key, so he can verify the signature on the certificate. As long as Bob trusts the CA, he also trusts that PK_A actually belongs to Alice" (Ferguson 2010, 275).

There is not one single CA, but rather several co-existing hierarchies of thousands of CAs. This creates problems of their own. If only one of the CA's self-signed root

[20] Alice of course might also decide not to trust Carol's key, if Bob is the only person, she trusts, that has signed it, i.e. she might require at least two (or more) trustworthy signatures before she trusts another key.

certificates is compromised or one CA does not do its job properly, this opens the window for attacks (Ferguson 2010, 284).[21]
A PKI is one of the best tools to solve the problem of key distribution in cryptography and it seems to be particularly suitable for our case, since we already have a centralized server, that could easily be used for key distribution. Nevertheless, we will have to take a close look at the details, since in our threat model we decided not to put (too much) trust in the server.

3.4.3 X.509 certificates

Certificates, of course, have been standardized and the most common standard are X.509 certificates (IETF 2008). An X.509 certificate consists at least of the following parts:

- Version Number
- Serial Number
- Signature Algorithm ID
- Issuer Name
- Validity period
- Subject name
- Subject Public Key Info
- Public Key Algorithm
- Subject Public Key
- Certificate Signature Algorithm
- Certificate Signature

Depending on the version of the certificate there may be additional entries. The entries in the certificate make sure the subjects name (which can be any kind of identifier, e.g. an email address or a company name) is tied to the public key, while also making some indications on how the authentication works (Ertel 2012, 130f).[22]

[21] For anyone interested in this problem and a proposal of a possible solution we recommend the paper by Kim et al. (2013).

[22] We will use X.509 version 3 certificates in this thesis.

3.4.4 The Socialist Millionaires' Protocol (SMP)

The Socialist Millionaires' Protocol (SMP) is a zero-knowledge-protocol[23]. In Off-the-Record Messaging (OTR) it is used for key verification in a synchronous communication session (OTR Development Team 2017). The premise is, that both parties share a secret. This secret can be relatively weak like a dictionary word, because the protocol takes place in real-time, so that an attacker has almost no time to perform a brute force attack on the secret.
In OTR once Alice and Bob have exchanged their public keys in an Authenticated Key Exchange (AKE)[24] either party can ask for key verification at any time. The fundamental idea is, that Alice and Bob share a secret p. This secret can be something they have agreed on, in a (previous) out-of-band conversation or any knowledge only the two of them share. Note that it is not unlikely that such a shared secret exists in the context of instant messaging and data sharing. Alice uses p to compute a secret value $x = h(P_A, P'_B, p)$ and Bob does the same computing $y = h(P'_A, P_B, p)$, where P_A P_B are Alice's and Bob's public keys and P'_A and P'_B are the public keys, they received from each other. This means $x = y$ is only true, if both $P_A = P'_A$ and $P_B = P'_B$ hold and they used the same secret p. The SMP allows Alice and Bob to check whether $x = y$ without leaking any other information to either party. If the statement is true, both know, that they share the same secret and use the same keys. Otherwise, they do not learn any information about the secret the other party used (Hallberg 2008).
Similar to a Diffie-Hellman Key Exchange Alice and Bob agree on a 1536-bit prime number n and a number g, that is (at best) a primitive root modulo n. These values can be publicly known. All exponentiations in the following are computed modulo n:

1. Alice picks three random numbers $a, a', r \neq 0$ and computes g^a and $g^{a'}$.
2. Alice sends Bob g^a and $g^{a'}$.
3. Bob verifies $g^a,\ g^{a'} \neq 1$.
4. Bob picks three random numbers $b, b', s \neq 0$ and computes g^b and $g^{b'}$.
5. Bob computes $g' = (g^a)^b$, $g'' = (g^{a'})^{b'}$, $P_b = g''^s$ and $Q_b = g^s g'^y$.
6. Bob sends Alice g^b, $g^{b'}$, P_b and Q_b.
7. Alice verifies $g^b,\ g^{b'} \neq 1$.
8. Alice computes $g' = (g^b)^a$, $g'' = (g^{b'})^{a'}$, $P_a = g''^r$, $Q_a = g^r g'^x$ and $R_a = (Q_a Q_b^{-1})^{a'}$.

[23] A zero-knowledge-protocol is a challenge-response-protocol that allows one party, the prover, to authenticate herself to another party, the verifier. The special characteristic of a zero-knowledge-protocol is that the verifier will get no information on the prover's secret. Nevertheless the verifier can be certain that the prover knows the secret. In the SMP the authentication is mutual (Lám 2016; Delfs and Knebl 2015, 118ff).

[24] The AKE is based on the Diffie-Hellman-Key Exchange. For details see OTR Development Team 2017.

9. Alice sends Bob P_a, Q_a and R_a.
10. Bob computes $R_b = (Q_a Q_b^{-1})^{b'}$ and $R_{ab} = R_a^{b'}$.
11. Bob verifies that $R_{ab} = P_a P_b^{-1}$.
12. Bob sends Alice R_b.
13. Alice computes $R_{ab} = R_b^{a'}$.
14. Alice verifies $R_{ab} = P_a P_b^{-1}$.

Note that g', g'' and R_{ab} are the keys negotiated in three intertwined Diffie-Hellman Key Exchanges. Therefore the following holds:

$$\begin{aligned} R_{ab} &= (Q_a Q_b^{-1})^{a'b'} \\ &= (g^r g'^x (g^s g'^y)^{-1})^{a'b'} \\ &= (g^{r-s} g'^{x-y})^{a'b'} \\ &= (g^{r-s} g^{ab(x-y)})^{a'b'} \\ &= g^{a'b'(r-s)} g^{a'b'ab(x-y)} \\ &= g''^{r-s} g^{a'b'ab(x-y)} \\ &= g''^{r} (g''^{s})^{-1} g^{a'b'ab(x-y)} \\ &= P_a P_b^{-1} g^{a'b'ab(x-y)} \end{aligned}$$

$R_{ab} = P_a P_b^{-1}$ implies $g^{a'b'ab(x-y)} = 1$ and because $a, a', b, b' \neq 0$, $x - y = 0 \Leftrightarrow x = y$ holds.
Therefore the final verifications convince both parties, that the other party has knowledge of the same secret value, derived from both public keys and the shared secret, and the keys are successfully verified (OTR Development Team 2017).

Concluding the section on cryptography, we state that cryptography is one of the most powerful technical tools to make a system secure and keep user data private. Anyway, it is important to keep in mind that no cryptographic system can provide absolute security. A brute force attack always has a small chance to succeed and further pitfalls lie in implementation errors that lead to vulnerabilities of the whole system. Most importantly it is important that cryptography by itself does not render any system secure, but rather the security of a system depends on the correct application and implementation of cryptographic methods. This includes the notion, that one of weakest links in any system is the user. Therefore the system should be designed such that the user cannot unwillingly compromise the security of the system.

4 Related work and existing client-side encrypted cloud storages

In this section we will first discuss the research that has been done in the field of client-side encrypted cloud storages. In the second part existing software will be

analyzed. The key question in both sections is how users can securely share data over a cloud storage without trading off usability.

4.1 Research on cryptographic cloud storages

In recent years a lot of research has been done on cloud storages and how to improve their security and privacy properties. In this section we will give a brief overview over the most relevant papers in the field. Some of them are discussed in detail in the corresponding sections.

Grolimund et al. (2006) propose a solution for efficient and secure key management in Sync and Share applications such as cloud storages. Their cryptographic tree structure, called "Cryptree" is the blueprint for most of today's client-side encrypted cloud storages. The authors introduce lazy re-encryption and separated read and write tree structures. Their Cryptree also contains so called "backlink keys" to access the metadata of parent folders. From our perspective this makes certain operations unnecessarily complicated and also represents a threat to user privacy, because a user might get information on data in parent folders that has not actually been shared with her. This is why we present a solution that works without backlink keys. We trust Grolimund et al. in their evaluation and believe that our own system will at least achieve the same performance and better in some cases due to the removal of backlink keys.

Lám, Szebeni, and Buttyán (2012b) present Tresorium, which is the basis for the CSP Tresorit (see section 4.2.5). They base their approach on Cryptree. In contrast to our design Tresorium does not allow to distinguish between read and write access when sharing data. Their approach for key management and distribution scales better when sharing data with large groups, because their focus is on a mechanism to negotiate and distribute group keys. This mechanism is further elaborated in Lám, Szebeni, and Buttyán 2012a.

Zwattendorfer et al. (2013) resolve the problem of key distribution in client-side encrypted cloud storage by using a PKI, based on the Austrian Citizen Card. In their design users retrieve certificates, issued by the Austrian state, from a central LDAP directory. One advantage of this architecture is that encryption is always hardware based and it is less likely that the private key stored in the Smart Card will be compromised. Unfortunately this requires both access to a Smart Card and a card reader, which is not very usable in particular on mobile devices and in countries, where states do not offer this infrastructure. Furthermore the design basically just moves trust from the CSP to the Austrian state, because the state could have made a copy of the private key before handing out the Citizen Card or simply provide fake certificates to perform a man in the middle attack as described in section 8.1. Because of these problems we discard their solution for our application scenario.

Körner and Walter (2014) also propose a solution to the problem of public key distribution and verification in client-side encrypted cloud storages by using QR-Codes as a carrier medium. They encode the complete public key in a QR-Code, in order to use it for distribution. For us this solution is not considered usable enough, because it impedes sharing, when users have not physically met before, which is why we make leverage of the server for key distribution. Yet, inspired by their paper, we

offer a QR-Code mechanism to verify other users' certificates (see section 8.2).
Zhao et al. (2010) introduce a progressive encryption scheme which allows data to be encrypted multiple times by clients in a row and be decrypted in only a single operation. Unlike our design such a scheme allows multiple users to work on documents simultaneously. However, there are some limitations, the authors themselves mention. Most notably it creates a large computational overhead and it requires cooperation from the CSP that needs to share its own private key with all users a file is shared with.[25] Due to these limitations we chose not to implement their mechanism, although it provides interesting properties for collaborative and simultaneous work on documents.
Kamara and Lauter (2010) present a paper on cryptographic cloud storage, that uses modern functional cryptography to enforce access policies on the data stored and shared in the cloud. While these policies mostly aim at a corporate environment, the functional cryptography used by the authors also enables searchable encryption. However, the authors do not implement their proposed system, because certain obstacles still have to be overcome. While some of the properties presented are very interesting, more research has to be done on the functional encryption schemes, before the solution becomes practically and efficiently usable in a real world application.
Wilson and Ateniese (2014) evaluate existing CSPs that claim to provide secure end-to-end encrypted cloud storage. Similar to our evaluation in section 4.2 they find that all existing CSPs lack certain security properties. In particular they point out that the CSP should not be trusted as the CA, if a PKI is used (cf. section 8.1). The authors only analyze a limited number of systems excluding for example Nextcloud. Since their paper is from 2014 they also cannot consider all recent changes in the evaluated systems, which is why we decided to do our own evaluation in the following section.
In recent years there have been different approaches to improve on the shortcomings of PKIs and to resolve the problem of end user public key verification in particular. One such approach is CONIKS (Melara et al. 2015), which allows users to monitor their key bindings and thus make sure no false public keys can be published. This removes the need for a trusted CA. However, a traditional PKI, in which the server acts as a CA and key distributor is better suited for the architecture of our system. Therefore, we do not implement CONIKS, but provide custom key verification by QR-Code scanning and the Socialist Millionaires' Protocol.
In our literature research we came across many more papers that try to resolve the problem of client-side encrypted cloud storage (cf. Gadhavi et al. 2016, Rewagad and Pawar 2013, Kaur and Singh 2013 and Kumar et al. 2012). These are not all presented in detail here, because they either aim at different security goals than we do or do not provide any contributions, that we have not talked about in the papers previously mentioned.

[25] Still, the cloud provider does not get access to the plaintext of the shared data.

4.2 Comparison of existing client-side encrypted cloud storage systems

In this section we will compare existing cloud storage systems that implement client-side encryption. Most of these systems are not open source, which means we have to base our analysis on the descriptions the companies provide in their white papers and other publications. Since these sources are also used for marketing and often lack important technical details, they should be carefully assessed. The goal of our comparison is not to come to a final conclusion on how secure a particular system is in practice or to make recommendations about which systems to use, but rather to analyze, how our requirements, described in the previous sections can be met and how different mechanisms are sensibly combined in existing encrypted cloud storages. That means the objective is to analyze which consequences each mechanism has for the functionality, security and usability of the entire system.
In our comparison we will focus on cloud storage systems that have a focus on private users and small-scale businesses. This means we will not explain in detail the solutions offered for the deployment in large- and medium-scale businesses,[26] but focus on systems that are available to end users. It also implies that our focus is not on the integration with existing (business) applications or scalability. In our analysis we will therefore describe the functionality provided by a system, describe the cryptographic methods involved with a particular focus on key management, and summarize the pros and cons of the solutions regarding security and usability.

4.2.1 Spideroak

SpiderOak has become famous as a Dropbox alternative recommended by Edward Snowden in 2013 (Kiss 2014). SpiderOak then claimed to be "Zero-Knowledge" and later changed the term to "No-Knowledge", since the system does provide end-to-end encryption, but the login mechanism does not fulfill the requirements of a cryptographic zero-knowledge-protocol (SpiderOak 2017c).
Functionality: SpiderOak is a backup software. Its main objective is to provide a client (for all common operating systems) that automates the process of frequently uploading local backups to the cloud. The user can decide whether to backup just one file, some folders or the complete local hard drive. Due to its focus on backups SpiderOak does not allow to share files with other users, but it is possible to make files and even directories publicly available via a web link. While these links still require a login password to be accessed, the files themselves are not stored end-to-end encrypted on the server, but only protected with server-side encryption (SpiderOak 2017b).
Security: SpiderOak uses hybrid encryption. The data is encrypted on the local machine before being uploaded to the SpiderOak servers. Each file and folder has its own key. Files and file metadata are encrypted using AES-256. Integrity and authenticity are ensured using HMAC-SHA256. The keys are stored on the SpiderOak servers encrypted with the user password using PBKDF2 with 16000

[26] Anyone interested in such systems might take a look at Syncplicity (2015) and FTAPI (T. 2014).

rounds of SHA256 and a 32 byte salt. Spideroak calls this privacy environment "No-Knowledge" and claims that the "server sees only a sequentially numbered series of data containers" (SpiderOak 2017a). The fact, that the keys are stored on the servers alongside the data and only encrypted with the user password makes key exchange between devices a trivial problem. The user simply has to login providing her username and password, thus synchronizing keys with the existing account. SpiderOak also uses data deduplication[27] across the data of a single customer not across customers. It is possible to do this locally and detect duplicate files by creating an encrypted database of hash values of the plaintext of every file, but SpiderOak unfortunately does not provide any information on how data deduplication is actually realized (SpiderOak 2017b, 2017a).

Usability and Conclusion: End-to-end encryption in SpiderOak is completely transparent, since the user only has to provide her username and password to access her account. SpiderOak is very easy to use from all common platforms and also has a web panel. The web panel, however, is only accessible after accepting a disclaimer, that informs about how the No-Knowledge policy is violated by accessing one's account from the web panel.[28] Generally SpiderOak protects the confidentiality and integrity of the reliably stored data. However the security largely depends on the password chosen by the user. The major disadvantage SpiderOak suffers from, compared to other solutions, is that it is not designed to share the files stored in the application. As a simple backup storage SpiderOak provides a secure and easy-to-use solution.

4.2.2 Cryptomator

Cryptomator is not a CSP, but an application that is run locally and enables users to store their data protected by end-to-end encryption on a regular CSP's servers (e.g. Dropbox, Google Drive, Nextcloud).[29]

Functionality: While Cryptomator claims to implement fully "transparent encryption", it actually breaks the sharing functionality of the cloud storage platforms it is based on. It only allows users to upload their encrypted files and access them, but it lacks a mechanism to exchange keys between users and thus makes it impossible to share encrypted files with other users. Although providing more or less the same functionality, the main difference to SpiderOak is, that Cryptomator creates a vault, that can be accessed just like an external drive such as a USB-stick (Cryptomator 2017a).

Security: Cryptomator realizes encryption similar to SpiderOak. Since it is only run locally, every encryption operation is performed locally before the data is stored in the cloud. Upon the creation of a vault, two 256 bit master keys are created for

[27]Data deduplication means that the exact same file is never uploaded and stored more than once in the cloud. Hash values of the files are compared to detect duplicate files. Instead of uploading the duplicate, only a link is created to the file that is already stored on the servers. This allows to save bandwidth and storage.

[28]The decryption in the web panel takes place in a JavaScript application, that is delivered by the server and should not be trusted.

[29]There exist similar applications, like Boxcryptor, but since only Cryptomator is open-source, we only analyze Cryptomator in this paper.

encryption/decryption (AES) and for authentication (HMAC). Upon encryption of a file or folder a unique AES-256 key is created and used for the encryption. This key in return is encrypted with the master key and stored alongside the encrypted file. The master keys are also encrypted with the user password using scrypt and stored in the root directory of the vault. To unlock the vault a user has to provide her password, which decrypts the two master keys and allows her to encrypt and decrypt files. Cryptomator encrypts every file and its metadata automatically, before it is uploaded to the cloud. The upload is taken care of by the actual client application provided by the CSP. Cryptomator decrypts every file automatically when accessed (Cryptomator 2017b).
Usability and Conclusion: Cryptomator provides good usability. Since it can be assumed that users know how to access a USB-drive, users will also be able to use Cryptomator. Data encrypted with Cryptomator is not accessible via a web interface. Due to the use of scrypt and two different keys for HMAC and AES-256, the authenticity, integrity and confidentiality of the data is protected slightly stronger than with SpiderOak. However, a compromised password will always lead to a full compromise of the user's vault, for anyone with access to the user's data in the cloud storage. The lack of a mechanism to share files, might detain potential users from using its services.

4.2.3 SafeMonk and Sookasa

SafeMonk was the first Dropbox Add-on that provided client-side encryption. In 2015 the developer, SafeNet, decided not to continue with the project and SafeMonk was merged into its former competitor Sookasa (Scherschel 2015). Since there is not much difference between both services, we will only describe SafeMonk.[30]
Functionality: SafeMonk works on top of Dropbox, somewhat similar to the way Cryptomator does. However, there are certain differences in the key management, that allow to preserve the sharing functionality of Dropbox. Like Cryptomator SafeMonk creates a vault, that is a special encrypted folder inside the Dropbox folder. The content of this folder will always be encrypted (Raymond 2015).
Security: In SafeMonk all cryptographic operations are performed by the client. Unlike the solutions presented so far in SafeMonk the structure of the encrypted directories is reflected in the key structure, i.e. SafeMonk implements a cryptographic tree structure as presented by Grolimund et al. (2006). In SafeMonk every file has a unique encryption key (AES-256). This key is encrypted with the encryption key of the parent directory, which in turn is encrypted with the encryption key from its own parent directory. This goes on to the root directory, the SafeMonk vault, which also has an encryption key, but this key is only encrypted with the user's public RSA key. Organizing encryption in this manner creates a tree structure, where each node contains a key that decrypts its children's keys. So every key can be restored to plaintext, if and only if the user owns the correct private RSA key to decrypt the root directory encryption key. The user's private RSA key is stored on the server encrypted with the user's password.[31] The tree structure has the purpose to allow

[30]There also is a white paper for Sookasa (2017) available.
[31]Unfortunately we did not find out, which algorithm is used to encrypt the private key.

users to share files and folders as they like with no need for re-encryption, because a simple share of the folder key will allow the user to access all of its content. This mechanism makes sure other users only gain access to the files and folders they are supposed to access, while the rest of the data remains confidential. SafeMonk posts the user's public key to the SafeMonk server. SafeMonk claims that keys are only stored in the client application (with the exception of public keys) (SafeMonk 2013). However, it is possible to exchange keys between devices, by simply logging in to the Dropbox and SafeMonk accounts with the user's passwords. It is also possible to access the encrypted data from the SafeMonk web interface with mere knowledge of the password. So, what they probably really do, is storing all user keys on their servers encrypted with the private key, which is also stored on the servers protected by the user's password (Raymond 2015). The security of this mechanism is not completely trustworthy, because of this unclear information policy and they do not state which password-based key derivation function is used to protect the private key.
In case a user wants to share a file, she shares the file as usual using Dropbox. Upon receiving the share the other user, who apart from Dropbox also has to run the SafeMonk client, requests the decryption key for that folder from the SafeMonk server. The SafeMonk servers ask the first user to confirm the share. Her client will then retrieve the public key from the SafeMonk server and use it to encrypt the folder key. This encrypted folder key is sent back to the server and from there to the other user, who has now gained access to the folder (SafeMonk 2013).
SafeMonk does not implement further measures to ensure the integrity and authenticity of the files stored in the cloud. Its focus is on confidentiality only.
Usability and Conclusion: While the tree structure that allows very flexible and transparent sharing, is a great plus to the functionality of SafeMonk, it has some issues, that could be resolved better. First SafeMonk uses its own servers for key exchange, thus creating a second infrastructure next to Dropbox, that could have been avoided with better integration. This infrastructure does not provide any additional security. In the described key exchange between Alice and Bob it is notable that the SafeMonk server could always act as a man in the middle providing fake keys. This puts a lot of trust in the SafeMonk servers, since there is no mechanism to verify Bob's public key. Although the public key distribution is not secure, it still requires some simultaneous interaction from the users without a gain in security.
In general it can be stated, that the tree structure used for key management is very useful, while the mechanisms for public key distribution are neither completely secure nor provide good usability. Another problem is that SafeMonk does not always explain how things are done, e.g. where the keys are really stored. Under these circumstances it is doubtful, whether the code delivered by the SafeMonk web panel can be trusted not to compromise the user's password and secret keys.

4.2.4 Sync

Sync is a commercial CSP that focuses on privacy (at least in their marketing): "'Zero-knowledge' means we absolutely cannot access the encrypted data stored on our servers. Your data is completely safeguarded from unauthorized access, which

is the only way you can completely trust the cloud. Protecting your right to privacy is our passion" (Sync 2017a).
Functionality: Sync provides functionality similar to that of Dropbox. It is possible to use Sync to backup files, synchronize files between devices or share them with others. It supports all common operating systems and also is accessible via a web panel. Sync also allows users to share data via web links.
Security: Sync's security is based on an RSA-2048 key pair, that is used to decrypt all other keys. The private RSA key is stored on the Sync server encrypted with the user's password using PBKDF2. The initial generation of the key pair can be done by the client or inside the web panel. Although Sync claims to be "Zero-Knowledge" it sends a BCRYPT hash value of the user password and a salt (fetched from the server) to the server in order to authenticate the user and be able to access the secret keys. BCRYPT is a one-way cryptographic hash function and therefore technically does does not fulfill the criteria of a zero-knowledge-protocol.[32] In Sync each file is encrypted with a unique AES-256 key that is stored on the server encrypted with the users private RSA-key. The metadata is encrypted with AES-256 using a key that is derived from the user's password with PBKDF2. When a user creates a share, a unique share key is generated and encrypted with public RSA keys of all users the data is shared with. This share key is used to encrypt all file keys (Key Wrapping). If a user creates a secure link to share files, the shared files are encrypted with the share password using PBKDF2 and AES-256 and the password is added to the URL after a hash tag (fragment identifier), so that it is never transmitted to the server, but can be used locally by the JavaScript code to derive the decryption key to unlock the share.
Sync can be accessed via a web panel. In this case JavaScript Code is delivered by the server and used for key generation and all other cryptographic operations. Sync is aware that this means putting a certain level of trust in the server and has thus made the JavaScript code for the web panel publicly available, so everyone can verify that no secrets (password or keys) are leaked to the server (Sync 2017b). However, when the actually code is delivered, no integrity check is performed in the browser, so that the publication mostly must be seen as a demonstration of good will by Sync, since it is not enforced that the delivered code is identical with the code Sync published.
Usability and Conclusion: Sync provides good usability, since the encryption is completely transparent. It also uses modern cryptography and can therefore in general be considered secure. Sync has the same problem like all services that rely on the user's choice of a good password. Because a weak password will make it relatively easy for the attacker to gain access to the whole account. Although the key management used in Sync allows users to share data flexibly, a lot of keys need to be wrapped each time a folder is shared. This task can be sped up by a cryptographic tree structure, that would also save bandwidth. One of the biggest flaws Sync has security-wise is the lack of a mechanism to authenticate the users, that the data is shared with. There is no way to verify other users' public keys and one has to completely trust the server in providing the correct keys.

[32]The server could confirm an assumption on what the user's password is by calculating the cryptographic hash value for that password.

4.2.5 Tresorit

Tresorit is similar to Sync, but takes some extra measures to ensure security. It does not only protect the confidentiality of the data users upload to the cloud, but also the integrity.
Functionality: Like Sync Tresorit's main services are file synchronization across devices and sharing data with other users. This includes data backups. Tresorit also allows users to access all their data from a web panel and share files via web link.
Security: In Tresorit all cryptographic operations are performed in the client. Each user has two pairs of private RSA-4096 keys, one for encryption (wrapping the symmetric AES-256 root key) and one for authentication and integrity checks (based on HMAC). These private keys are stored in the user's root directory on the server encrypted with the user's password using PBKDF2 with HMAC-SHA1. Before the upload each file is encrypted with a unique AES-256 key. To protect the integrity of the files Tresorit uses HMAC-SHA512. Each upload and modification of a file is also authenticated with the user's digital signature using an RSA-4096 key and SHA512 as a hash function. To store the keys Tresorit uses a tree structure similar to that of SafeMonk, i.e. each file key is encrypted with the key of the parent folder. Unlike SafeMonk, Tresorit stores the keys alongside the files itself in the same folder and uses no extra infrastructure for key management. Using this key structure, sharing is realized by making the decryption key of the shared folder available to the users it is shared with. When sharing with a group of users there is one pre-master secret for each user granted access to the files. To unlock a share the user's private RSA key is used to decrypt this user's pre-master secret. The resulting key is used to calculate the AES-256 key of the shared directory (which unlocks the root key for that share) by computing the HMAC of the sharing users' public key certificates.[33] The certificates used are issued by Tresorit itself and are X.509 certificates, that contain the user's email address as an identifier. This means Tresorit runs its own PKI, in which it acts as the CA. However the authentication mechanism for another user's key is more sophisticated than in a standard PKI: When Alice wants to share a file with Bob, she invites him to the share by providing Bob's email address. Tresorit then sends an email to Bob, that contains a 256 bit random secret. When Bob accepts the invitation in Tresorit, he has to provide this secret in a challenge-response protocol. Tresorit sends both, Alice and Bob, the certificate and the public key of their counterpart, which is supposed to guarantee that they really have the correct certificate for the user identified by the email address.
Considering that the secret is sent by the server itself, it is not a suitable measure to prevent a man in the middle attack with collaboration of the server. However, if Alice is not certain, that this is secure, she can use an out-of-band channel to agree on a password with Bob and both will have to provide this password, in order to mutually verify their certificates. Unfortunately Tresorit does not offer more details on and how the password is used to verify the counterpart's certificate. Therefore we cannot trust the key verification in Tresorit. In our design we will use a similar and well documented scheme, the Socialist Millionaire's Protocol.

[33] Note, that there is an alternative version, which uses a Diffie-Hellman Key Exchange instead of pre-master secrets (Tresorit 2017b). The group key negotiation Tresorit uses has been described in detail in Lám, Szebeni, and Buttyán 2012a.

Tresorit allows users to revoke shares. In such case Tresorit deletes the user from the Access-Control-List (ACL) of this share and re-encrypts the share's root directory. This immediately impedes access to the root directory by the revoked user. In order to safe computing power, the rest of the data is only re-encrypted, as it is changed (lazy re-encryption). This makes sure a revoked user will see no further changes made to a document she could formerly access. When a user logs in, a challenge-response protocol is applied. The server sends the user a challenge and the 160 bit salt. The user uses PBKDF2 to compute a key from the password and the salt. She calculates the HMAC of the challenge with that key. The result is the response which she sends back to server. The server has stored the key derived from the password and salt and also computes the HMAC of the challenge. If the result matches the response, the user is granted access to her account (Tresorit 2017b, 2017a).
Usability and Conclusion: Tresorit provides all basic cloud storage functionality like file synchronization, sharing and access via a web panel. It is the most secure of the presented solutions. This is due to its application of modern cryptography and its neat key management. It also has the most elaborated key distribution and is the only solution that puts a lot of effort in ensuring data integrity (using separate keys for encryption and integrity checks). However, its security is based on the user's choice of password, as the password unlocks everything, but users usually choose weak passwords. In our own system we will resolve this problem by enforcing strong random passwords, that will also enable cross-device key exchange.
When using Tresorit's web panel the user has to trust the server not to deliver malicious JavaScript code, that is used for the cryptographic operations. But since it is closed-source software users have to put the same trust in the client software. However, the most important problem regarding Tresorit is, that it does not offer a well-proven key verification mechanism, which means a man in the middle attack by the Tresorit servers (which act as a CA) might always be possible (cf. 8.1).

4.2.6 Nextcloud

The ownCloud fork Nextcloud presented its design for end-to-end encryption and a proof of work Android client in September 2017. It is the first cloud software, that is open-source and therefore self-hostable, that provides client-side encryption.[34]
Functionality: Nextcloud is a cloud storage software that can be self-hosted or deployed in a public cloud. It provides the basic sync and share functionality, i.e. users can upload their files, sync them between devices and share files with other users and user groups. It also offer a great number of additional applications such as calendars, password managers, audio/video calls etc. (Nextcloud 2017b). Nextcloud after the integration of end-to-end encryption will only support end-to-end encryption for new empty directories. It means while encrypted and not encrypted folders may live in the same account, an existing unencrypted folder cannot be encrypted. End-to-end encrypted folders are not displayed and cannot be accessed via the web interface or in older versions of the client that do not support

[34]Seafile is also open-source software and generally performs encryption operations on the client, but for some use cases the password is sent to server and used to derive the key (Seafile 2017), which is why we do not consider it in this section.

end-to-end encryption.
Security: Nextcloud allows users to create encrypted folders. These folders can be shared with other users. For the encryption Nextcloud uses AES-128 with a unique key for each file. It encrypts all metadata (in a JSON object) and all file keys with an AES-128 share key, thus not giving away any information apart from the number of files contained in the folder. The AES encryption key is encrypted with the public keys of all users, that are supposed to get access. When setting up her account for end-to-end encryption the user generates an RSA key pair and retrieves a X.509 certificate from the server, that acts as a CA in the PKI used for key verification. The user's private key is encrypted with PBKDF2 with HMAC-SHA1 using 12 randomly chosen words from a 2048 words list as a password. When the user sets up the client on a new device, it downloads the public key certificate for her user ID and the corresponding encrypted private key. If they match, the user can decrypt the private key with the 12 word password and store it on the device. When a user wants to share a folder with other users, she first has to get their certificates from the server and make sure they are signed by the the CA. When a user' access to an existing share is revoked, lazy re-encryption is applied (Nextcloud 2017a).
Usability and Conclusion: Nextcloud is the first open-source cloud storage software that provides client-side encryption. However, there are some pitfalls. Regarding usability the most significant disadvantage is probably that Nextcloud is inflexible with sharing. Users need to create special folders for encryption and cannot just encrypt and share existing folders. Also it is impossible to re-share a subfolder of an existing encrypted share. Nextcloud could have benefited a lot from a cryptographic tree structure. However it provides a secure and user friendly mechanism for cross-device key exchange. Currently key verification is based completely on the trust in the CA and users are not allowed to revoke and change their private keys, which means that there is no way to recover from a compromised private key. Generally the end-to-end encryption in Nextcloud is a good approach, that is suitable for many use cases, but it lacks flexibility and proper key verification and key revocation mechanisms. In Nextcloud client-side encryption is not provisioned to be supported by the web panel due to the security risks this represents.

Summarizing our analysis of existing client-side encrypted cloud storage solutions it is worth pointing out that there is a number of systems available that use secure cryptographic algorithms such as AES and RSA/Elliptic Curves. These systems hybrid encryption to achieve secure and efficient systems. Unfortunately all of these systems suffer from severe limitations, e.g. Spideroak and Cryptomator do not allow users to share encrypted data. Sync, Nextcloud and Tresorit rely on a PKI managed by the CSP in order to offer encrypted sharing. All three systems suffer from the lack of a secure certificate verification mechanism to prevent man in the middle attacks. This even is the case for Tresorit, which also suffers from weak protection of the user's private key. Nextcloud, on the other hand resolves this problem by forcing a 12 word random generated passphrase to protect the user's private key, but Nextcloud's PKI does not provide essential mechanisms such as certificate revocation in case of a private key compromise. In the following sections we will describe our own design

in detail and provide new mechanisms or combine the so far presented mechanism in a new manner, in order to achieve a secure client-side encrypted cloud storage.

5 Why client-side encrypted cloud storage should not provide web access

So far we have discussed how data can be stored, accessed and shared securely via a client application installed on the user's devices. In this section we will discuss, how encrypted data can be accessed via a web panel and why chose not to support this feature.

We made it a key requirement, that the client application is open-source, so that no back doors can be introduced without the knowledge of the user. The authenticity and integrity of the client application code can easily be checked with a digitally signature on the binaries. These mechanisms, that make it unlikely that a client software was modified and malicious code injected without the knowledge of the user, do not exist for the (JavaScript) applications needed to provide a full functioning web panel.[35]

Given our current design it would be easy to implement a JavaScript application that retrieves the encrypted private key from the server and provided the user's passphrase decrypts the user's files in the web browser. There are two problems with this approach. First web browsers are generally not a very secure environment and more importantly using JavaScript code delivered by the server to handle the private encryption key means the user has to trust the server not to deliver any malicious code, that will leak information about the data or the encryption keys to the server. For anyone with access to the server such a modification of the code is quite easy, which means such a web application actually is a back door by design. Since this problem is well known, Sync (2017b) decided to publish the source code for the web panel while Nextcloud (2017a) decided not to provide a web interface at all. In our threat model we have made the decision not to trust in the server. Therefore we cannot provide full web access to encrypted accounts.

In recent years interesting approaches were made to find a solution for the problem of JavaScript Code integrity, which is even more relevant when the connection is not encrypted with SSL/TLS.[36] A mechanism for the web browser to check the integrity of the code is needed, in order to avoid the execution of unverified JavaScript code. In 2016 a mechanism to introduce Subresource Identity has been introduced and is supported by the most common web browsers (W3C 2016). Subresource integrity allows developers to specify a cryptographic hash for third party resources in order to protect the integrity. However, when the server itself is not trusted to deliver the correct code, this mechanism is useless, because the browser does not have a way to

[35] Obviously most users will not really analyze the code, but hopefully the community will. Still, this can never provide absolute security. For example in February 2016 the website of Linux Mint was hacked and the download links were modified to point to a malicious version of the operating system. The integrity checksums on the website were also modified, so that users would not easily detect the hack. The problem was fixed within hours (Linux Mint 2016).

[36] In this scenario anyone with access to the connection between server and client could modify the JavaScript.

get a cryptographic hash from the server. So there is more research required on how to ensure the integrity of code delivered from the server in critical scenarios like the handling of confidential data.
So far we only know of one solution that actually works: the user installs an extension in her local browser. This extension has either a fixed integrity checksum for the code or it fetches the integrity checksum from a trusted server, like a Github repository where the complete code is published. The extension will check the code and if it does not pass, display a warning or prevent the execution. A proof of concept of such an extension already exists for the software PrivateBin (2016).
Unfortunately this is far from perfect, because the user always has to install an extension each time she wants to access the web interface from a new browser or device. If this is the case a user also could simply install the Desktop or mobile client application, because the main point of having the web interface is, being able to access one's data from any machine right away without installing any software. It might also be hard to keep the extension up to date, if the JavaScript code from the server frequently changes. So the extension might be a hindrance to continuous development of the web application. Most importantly it cannot be ensured users only access the web interface after they have installed the browser extension. It is not unlikely that users would access the web interface without installing the extension because it is more convenient and the web interface would work without the extension or could at least be modified in such a manner by a malicious CSP. For these reasons we have decided not to use a web extension to ensure code integrity and not to provide web access at all.[37]

6 Key management

In this section we will propose an architecture for a client-side cloud storage that allows us to enforce our security goals, especially confidentiality, authenticity and integrity of the user's data. At the same time it is flexible and does not break the core Sync and Share functionality of cloud storages. We will not describe any details of basic functionality of cloud storages, but only focus on the aspects relevant for the security of the client-side encryption, that is mainly handling of cryptographic operation and key material.
In server-side encryption the CSP always has access to the keys. So the crucial question of key management is who gets access to the keys. This thesis presents a design that exclusively allows authorized users to access the file encryption keys. This is what we call client-side encryption. Apart from the question, who can access the encryption keys, we organize encryption in a way, that allows the system to be efficient and does not sacrifice performance. Also, it preserves the original functionality and usability as best as possible.

[37] As a security-usability trade-off it might be an option to allow users to share particular folders via a web link. This would easily be feasible given the cryptographic tree structure and the RESTful API we use. Because of the previously mentioned reasons we did not include it.

6.1 Architecture

In this section we describe the cryptographic tree structure used to organize encryption of files and folders, i.e. how keys depend on one another and where they are stored. In the following section we will describe how operations are performed.
We propose a cryptographic tree structure, similar to what has been used in practice in systems like SafeMonk and Tresorit. The design was first published by Grolimund et al. (2006) under the name "Cryptree", of which our design is an adaptation. Cryptree's main benefits are simplicity, efficiency and intuitive semantics. Note that there exist different variants of Cryptree, e.g. the inventors of Tresorit describe an early version of how they implemented it in their system "Tresorium" (Lám, Szebeni, and Buttyán 2012b) and Mathieu (Bourrier 2016) uses it in his draft of client-side encryption for ownCloud under the working title "ownCrypt". Our own architecture is inspired by those approaches, although there are some disadvantages to both Cryptree and Tresorium: Cryptree overcomplicates key management by introducing a back link key structure for "Upward Inheritance of Access Rights". This allows users to access the metadata of ancestor folders and turns the cleaning operation and the revocation operations into rather complicated procedures (see 6.2). Tresorium on the other hand does not allow to differentiate between read and write access[38] and provides only one "tresor", a shared folder, per group. The ownCrypt draft does not foresee to use lazy re-encryption and is therefore not optimal for frequent changes of access rights. The architecture presented here is designed to provide maximum flexibility and performance, while protecting the confidentiality, integrity and authenticity of all data stored in the cloud.
Since we want users to interact with the system as intuitively as possible leveraging the folder structure is an obvious choice. Our design achieves the following characteristics:

1. Sharing a folder also recursively grants access to all subfolders and files contained in it.

2. Sharing a folder does not grant access to any ancestor folders or sibling folders (that are contained in the same parent folder). This includes the parent folder's metadata.[39]

3. It is possible to distinguish between read and write access, where write access always includes read access but not vice versa.

4. File metadata is treated as confidential. It is considered acceptable to give away information on the size and number of files, but not other metadata (names, time of modification etc.) and of course not the file's content.

To achieve these goals we use key wrapping or as Grolimund et al. (2006) put it: "cryptographic links". This includes both symmetric and asymmetric links. To

[38] At least the distinction is not enforced cryptographically.

[39] This is different from what Grolimund et al. (2006) proposed, since we do not want users to have upward inheritance of access rights which would allow them to get information on the ancestor folders of a shared folder or file.

create a cryptographic link from $K_1 \rightarrow K_2$, K_2 is encrypted with K_1, thus anyone who has access to K_1 and the link, also has access to K_2. Cryptographic links can be interpreted as the edges of a graph, where the keys themselves are the nodes. In the following we will describe all data structures in detail. For every folder there is one read item and one write item. For every file there is one read item, one write item and the file ciphertext which is named with a random identifier. In these items the keys and the data needed to organize encryption are stored.

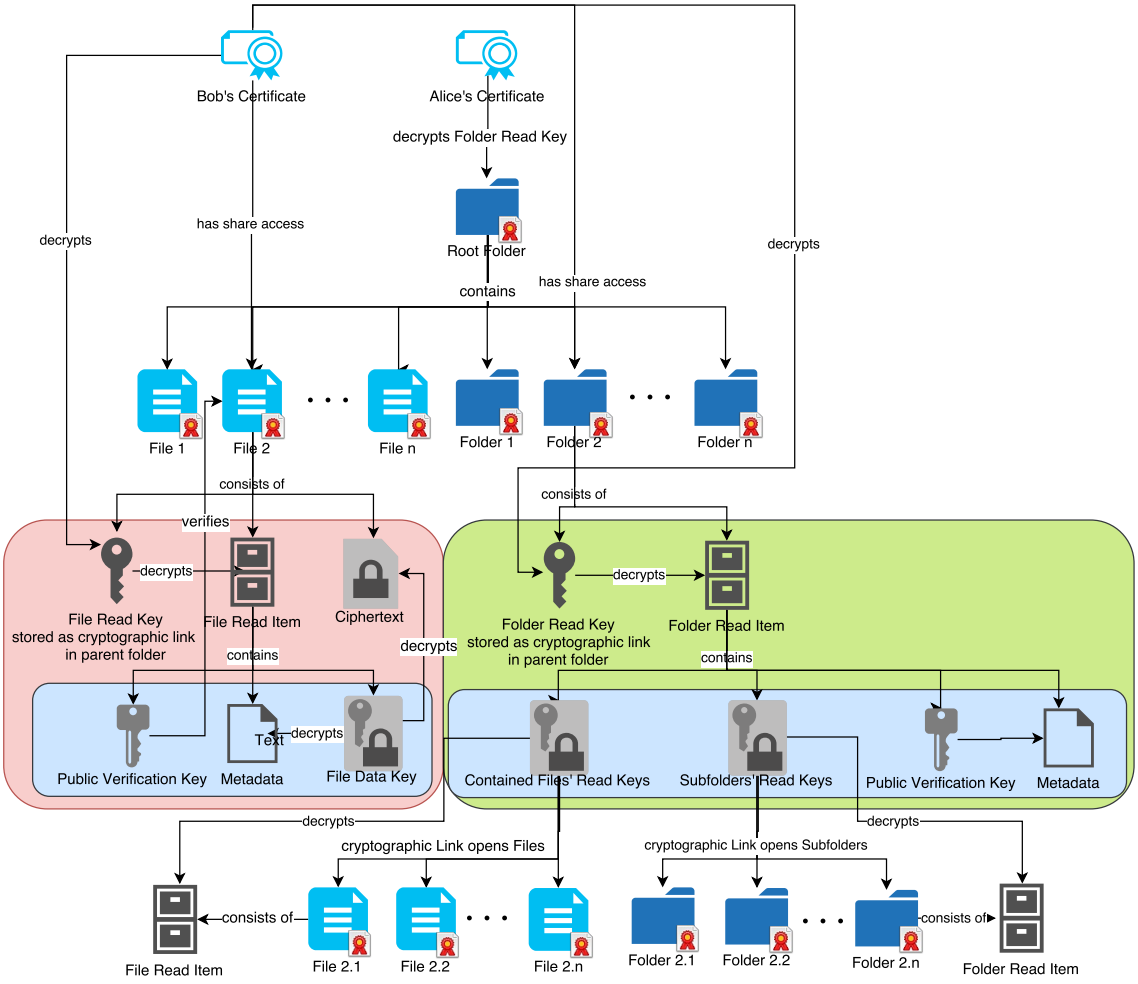

Figure 1: An example of a read cryptree
Red box: File 2. Green box: Folder 2. Blue boxes: the respective file/folder items.
Note the asymmetric cryptographic links from Bob's certificate to the File/Folder Read Keys are actually stored in the corresponding File/Folder Read Item (but of course not encrypted with the File/Folder Read Key).

In order to differentiate between read and write access, two different cryptrees are needed. First we describe the read access cryptree as shown in Figure 1. Every folder has a unique folder read key and a folder read item, in which the following information is stored:

- the metadata (name, creation, modification time, size etc.) of the folder (encrypted with the folder's read key)

- a cryptographic link to a unique subfolder read key for every subfolder that is used to encrypt the subfolder's metadata and create further cryptographic links
- a cryptographic link to a unique file read key for every file that is used to encrypt the file's metadata and the file data key
- a (public) verification key to verify the signature of all write operations to the folder
- an asymmetric cryptographic link for each user with read access to the folder from her public key to the folder read key (only needed for the user's root folder and shared folders, not for the subfolders, that are implicitly shared)

For each file there is the file's ciphertext a file read key and a file read item, in which the following information is stored:

- the file data key used to encrypt the file as stored on the server (encrypted with the file's read key)
- the metadata (name, dirty flag, modification time etc.) of the file (encrypted with the file's data key)
- a public verification key to verify the signature of all write operations
- an asymmetric cryptographic link for each user with read access to the file from her public key to the file's read key (only needed, if the file is explicitly shared with that user)

For write access there exists a separate, but similar tree structure shown in Figure 2. Public key cryptography is used to check the authenticity and integrity of write operations. The idea is that every change of the contents or metadata of files and folders has to be signed and only those with write access have the correct signature keys, but everyone with read access to a folder can check whether the signature is valid, because the verification keys are stored in read folder and file items. If not, the file is rolled back to the last valid version.
For each folder there is a folder write key and a folder write item, that contains the following information:

- a private signature key for all write operations (corresponding to the public verification key in the folder read item) encrypted with the folder write key
- a symmetric cryptographic link for every subfolder from the folder write key to the subfolder write key
- a symmetric cryptographic link for every file from the folder write key to the file write key
- an asymmetric cryptographic link for each user with write access to the folder from her public key to the folder write key (only needed for the root folder and shared folders, not for the subfolders, that are implicitly shared)

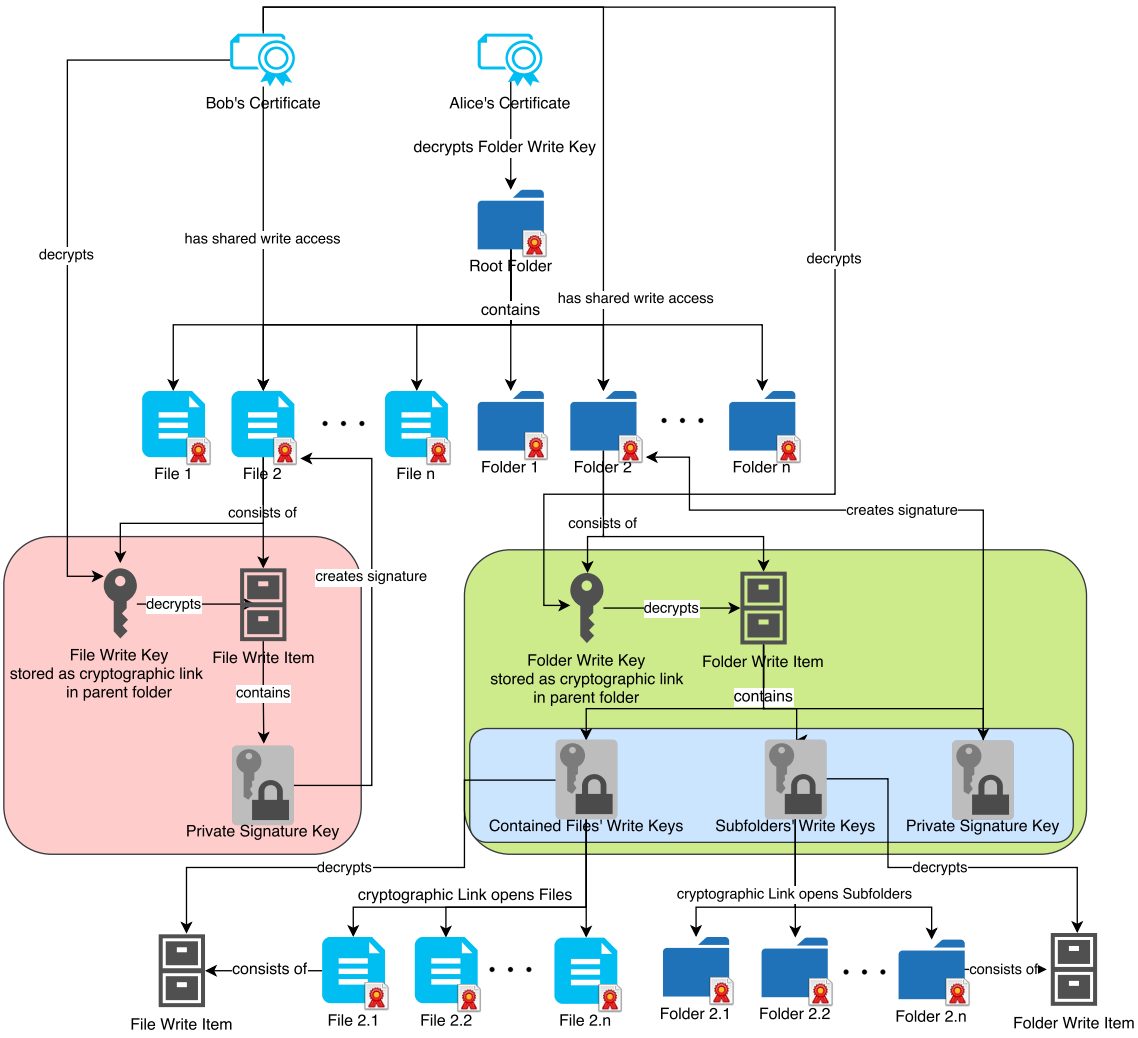

Figure 2: An example of a write cryptree
Red box: File 2. Green box: Folder 2. Blue box: the respective folder item.
Note the asymmetric cryptographic links from Bob's certificate to the File/Folder Read Keys are actually stored in the corresponding File/Folder Read Item (but of course not encrypted with the File/Folder Read Key).

For each file there is a file write key and a file write item that contains the following information:

- a private signature key for all write operations (corresponding to the public verification key in the file read item) encrypted with the file write key
- an asymmetric cryptographic link for every user with write access to the file from her public key to the file write key (only needed, if the file is explicitly shared with that user)

This structure allows granting read access independently from write access. Since a lot of signatures have to be checked, handling write access separately from read access can slow down the system significantly for certain operations, e.g. the revocation of write access requires a lot of large public/private key pairs to be re-generated and

a lot of write items have to be re-encrypted, because lazy revocation is not applicable here. So one should consider carefully, whether the separation of read and write access is really needed or whether everyone with read access, should automatically have write access which would cause a boost to performance of the system, since the write item tree could be left out completely. In that case faster HMACs, computed with the file and folder read keys, could be used instead of digital signatures, in order to protect data integrity and authenticity.
To achieve an acceptable performance our design has to avoid asymmetric cryptography, where possible. This is realized by giving each user only one private/public key pair, that serves as her identification and grants access to the keys used for data encryption. Because we use a tree structure that only consists of symmetric keys, the amount of public key cryptography needed is marginal. However, some additional public key cryptography is needed to make sure the data can be checked for integrity and authenticity. Public key cryptography in this scenario cannot be avoided since we do not only want users with write access to be able to perform the checks, but also all users with read access.
Apart from avoiding public key cryptography it is particularly important to avoid frequent re-encryption and changes of the encryption keys. The presented tree structure allows us to re-encrypt only those files that were actually changed reducing the use of bandwidth to a minimum, because we do not have to re-encrypt a complete folder and upload it again when a single file in it changes.
The proposed design guarantees the confidentiality of the user's data, when she revokes access for another user. Considering very dynamic user groups, i.e. with frequent changes of the group members, it is not efficient to substitute all the involved keys and re-encrypt all data immediately, as this might cause an enormous use of bandwidth (and computing operations of the client). The application of lazy re-encryption resolves this issue. Lazy re-encryption means, that when a user's access is revoked, she immediately loses access to the encryption keys, because the cryptographic link using her public key is removed from the folder/file read (and write) item. If the user still has access to the CSP and prior to the revocation made a local copy of the encryption keys, she can still access the files. This is considered acceptable, because anyone with a local copy of encryption keys will probably also have a copy of the actual data. So no additional information is leaked. Lazy re-encryption ensures that a user who has been revoked access to the share cannot keep track of any further changes. Therefore once a shared files changes, a new encryption key is generated and used to re-encrypt the new version of the file. Lazy re-encryption in our design is realized by marking files as dirty, when someone is revoked access and cleaning them before changes are uploaded. Because the design also aims to support, that users without write access to parent folders are able to clean the files contained in a folder, folder keys have to be kept clean at all times and an additional data key (used only to generate the file ciphertext) is needed in the file read item. File data keys are the only keys that can ever be marked dirty, because all other keys have to be renewed at once. So even with our realization of lazy re-encryption some folder and file keys have to be re-encrypted immediately, but the system avoids the re-encryption of files, that are usually much larger than 256 bit encryption keys. This saves a significant amount of bandwidth. Unfortunately

lazy re-encryption cannot be applied to write access rights, because nobody could distinguish between legitimate and illegitimate signatures. Therefore verification and signature keys are renewed immediately at the revocation of write access. This makes write access revocation one of the most costly operations in the proposed design.

6.2 Operations

In this section we will describe the operations concerning key management and cryptographic operations, but not the operations performed in the underlying cloud storage infrastructure. We will not explain trivial operations, that just work straight forward or can easily be composed of other basic operations.[40] Some operations require certain access rights, this is always assumed to be the case. For example moving a folder requires write access to both the old and the new parent folder. In the presented system granting and revoking access rights can be realized by anyone with write access to the shared folder/file.[41]
Every described operation is performed in the client application and handled as a transaction, i.e. it is either completely finished (commit) or not at all (rollback). The system does not support users to work on the files in parallel, because the server does not have any information on the files' content and can therefore not make use of versioning.

6.2.1 Adding a file

The client generates a new file read key, file write key, file data key and a signature and verification key pair for the file. The file read item is created containing the metadata, the file data key and the verification key using the read key for encryption where it applies. The read item is named with a random identifier. The file is encrypted with the file data key and and the ciphertext (also named with the file identifier) and the read item are stored in the cloud. A cryptographic link to that new read key is stored in the parent folder read item.[42] The file write item is created containing the signature key encrypted with the file write key. The file write item is stored in the cloud using the same identifier as the ciphertext and the file read item. In the parent folder write item a cryptographic link to the file write key is created.

6.2.2 Removing a file

To remove a file its read and write items and ciphertext of the file are deleted. The cryptographic links are removed from the parent folder read and write items.

[40] E.g. a file/folder is replaced by removing the old file/folder and adding the new one.

[41] Note that this means that it is theoretically possible to deny the owner access to its files. However, apart from what is cryptographically possible there will be an ACL, with which the CSP manages user privileges. So the owner might choose a configuration that restricts operations, that will render her unable to access her own data. Note, that in this thesis we refer to "ownership" not as the legal concept, but rather as having access to a key, data etc. In the context of cloud storages we call the user, who initially uploaded data, the "owner" of that data.

[42] The cryptographic link is actually the only place the file read key is stored until it is shared.

6.2.3 Moving a file

Cryptographic links in the new parent folder read and write items are created pointing to the read key and to the write key of the file respectively. The cryptographic links pointing to the read key and the write key are removed from the old parent folder read and write items. The file is marked dirty. [43]

6.2.4 Copying a file

First a physical copy of the file ciphertext and its read and write item is created in the old parent folder. Then this copy is moved to the new parent directory.

6.2.5 Updating a file

The file read and write items are cleaned (see below). To update a file the file data key is used to encrypt the new plaintext and the old ciphertext is replaced with it. The new metadata is encrypted with the file data key and the entry updated in the file read item.

6.2.6 Adding a folder

The client generates a new folder read key, folder write key and a signature and verification key pair for the folder. The folder read item is created containing the metadata and the verification key using the folder read key for encryption where it applies. The folder read item is named with a random identifier and stored in the cloud. A cryptographic link to that new folder read key is stored in the parent folder read item. The folder write item is created containing the signature key encrypted with the folder write key. The write item is stored in the cloud using the same identifier as the folder read item. In the parent folder write item a cryptographic link to the folder write key is created. Subfolders and contained files are added recursively.

6.2.7 Removing a folder

All subfolders and files are removed recursively from the folder. The folder read and write items are removed. The cryptographic links are removed from the parent folder read and write items.

6.2.8 Moving a folder

Cryptographic links in the new parent folder read and write items are created pointing to the folder read and write key respectively. The cryptographic links pointing to the folder read key and write key are removed from the old parent folder read and write items. The folder is marked dirty. [44]

[43] This makes sure that everyone with access to that particular file keeps it, while those with access to its old parent directory lose access. Everyone with access to the new parent folder gains access.

[44] This makes sure that everyone with access to that particular folder keeps it, while those with access to its old parent directory lose access. Everyone with access to the new parent folder gains

6.2.9 Updating a folder

To update a folder (e.g. rename it) the new metadata is encrypted with the folder read key and the metadata entry in the read item is replaced with it.[45]

6.2.10 Granting read access (Sharing)

To grant read access to a folder/file to another user a cryptographic link from the user's public key to the folder/file read key is added to the folder/file read item of the.

6.2.11 Granting write access (Sharing)

The user is granted read access to the folder/file, if she does not have read access already. To grant write access to a folder/file to another user a cryptographic link from the user's public key to the folder/file write key is added to the folder/file write item of the folder/file.

6.2.12 Revoking write access

When a user is revoked write access to a folder/file, the cryptographic link from her public key to the folder/file write key is removed from the folder/file write item. The write item of the folder/file is cleaned.

6.2.13 Revoking read access

The user's write access is revoked. The user's cryptographic link is removed from the folder/file read item and the folder/file is marked as dirty.

6.2.14 Mark a file as dirty

The file write item is cleaned. The dirty flag is set in the metadata in the read item. A new file read key is generated and the cryptographic link in the parent folder read item is updated to point to the new file read key. The new file read key is used to re-encrypt the file data key in the read item of the file. The asymmetric cryptographic links in the file read item are updated to point to the new file read key.

6.2.15 Mark a folder as dirty

The folder write item is cleaned. A new folder read key is generated and the cryptographic link in the parent folder read item is updated with that folder read key. The new folder read key is used to update the cryptographic links to the subfolders read keys and contained files read keys in the folder read item. The asymmetric

access.

[45]Note that folders actually only consist of metadata. So there is no need to apply actions recursively.

cryptographic links in the folder read item are updated to point to the new folder read key.[46]

6.2.16 Clean a file read item

If the file is not marked dirty, nothing needs to be done. Otherwise a new file data key is generated and used to re-encrypt the file and its metadata in the file read item. The file data key is updated in the file read item using the file read key to re-encrypt it. The dirty flag is removed from the file read item.

6.2.17 Clean a folder read item

Folders are always kept clean (see 6.2.15) and the contained files can be cleaned individually when needed.

6.2.18 Clean a file write item

A new signature/verification key pair and file write key are generated. The file write item is updated storing the new signature key encrypted with the new file write key and updating all cryptographic links for each user using the new file write key. The cryptographic link to the file write key is updated in the parent folder write item. The verification key is updated in the file read item.

6.2.19 Clean a folder write item

A new signature/verification key pair and folder write key are generated. The folder write item is updated storing the new signature key encrypted with the new folder write key and updating all cryptographic links for each user using the new folder write key. The cryptographic link to the folder write key is updated in the parent folder write item. The verification key is updated in the folder read item. All sub-folder write items and contained files write items are cleaned recursively.

In this section we presented an architecture and the most important file and folder operations for the key management in a client-side encrypted cloud storage. Our design enables flexible sharing and revocation. Contrary to existing solutions our design does not only allow to distinguish between read and write access, but it cryptographically enforces that users with share access to a folder cannot access any metadata of the parent folders. Most of the file and folder operations are similar to Grolimund et al. (2006), which is why we relinquish to do an evaluation of the cryptographic tree structure and instead focus on our key exchange and key verification mechanism presented in the following two sections.

[46] All keys are immediately updated, because otherwise updating a dirty folder would require access to the parent folder in order to update the cryptographic links in it. This would make it impossible to clean folders for users that do not have full access. For this reason we also use separate file read keys and file data keys.

7 Cross-device key exchange

In this section we will design the mechanism to exchange private/public key pairs between a user's devices. Our approach is inspired by the protocol described in Nextcloud 2017a. We point out that in our design, there will only be one private/public key pair (with the corresponding X.509 certificate) for each user. The alternative to this design would have been to use a unique private key for each of the user's devices. This would have made the revocation of a device easier, because only one device would have been needed to be revoked.[47] Usually users want to use the cloud storage to sync all their data between all their devices. Therefore this feature might easily be turned into a major inconvenience causing a lot of micromanagement of devices. A user that wants to grant another user access to a folder cannot be bothered to decide which of the other user's devices should be granted access. This is why we decided that each user only owns one private/public key pair.[48]
In order to get access to the encrypted files in the cloud on all of her devices the user only needs to transfer this key pair to these devices. To realize this task, we are going to use the server for both key distribution of public key certificates and storage of the user's encrypted private key. More precisely we use a PKI, in which the server acts as a CA. This allows us to distribute keys rather easily and also generate some basic trust, because it means in order to provide a fake public key or get access to a user's encrypted private key an attacker needs to control the the CA's private key used for issuing certificates. Since we do not want to trust the server completely, further mechanisms for key verification are explained in section 8.

7.1 Adding the initial device

When the client first registers for encryption it generates a RSA-4096 key pair. The client requests a X.509 certificate from the CA, the server, using the current username as an identifier and the freshly generated key pair. If there is no existing certificate for the client's username and the requesting client's username matches the provided username the server issues the certificate, stores it and sends it back to the client, i.e. the principle of Trust On First Use (TOFU) is applied. The client stores both the private key and the new certificate securely on the device. The X.509 certificate is publicly accessible on the server for any registered client.
In order to be able to recover from device loss and to transfer the keys to another device, in a second step the private key is encrypted and stored on the server: the client randomly generates a twelve word passphrase from the BIP-0039 word list, which contains 2048 words. This means there are $2048^{12} = 2^{132}$ uniformly distributed passphrases (given that strong random numbers are available). The client derives a key from this passphrase using PBKDF2 with HMAC-SHA256 and encrypts the private key with it, generating a X.509 private key. This encrypted private key is stored on the server, accessible only for a client with the current

[47] Also, it would have allowed users to define for every file and folder, which devices are granted access.

[48] Note that this key pair can get revoked and replaced by a new one.

username. The passphrase is displayed on the client and stored securely on the device. The passphrase is never transmitted.

7.2 Adding further devices

When the client registers a new device, it checks whether there is an existing X.509 certificate for the username. If not, see section 7.1. Else, the client retrieves the certificate from the server and verifies that the username in the certificate matches the one the client uses. Next the client downloads the corresponding X.509 private key from the server and asks the user to provide the passphrase to decrypt it. The client checks the decrypted private key really corresponds to the X.509 certificate it received. If so, it stores the private key, the passphrase and the X.509 certificate securely on the device.
Note this key exchange mechanism protects 1) confidentiality, because only a user with knowledge of the passphrase is able access the plaintext of the private key, 2) integrity, because the client checks whether the private key matches the X.509 certificate and 3) authenticity, because the received private key and corresponding X.509 certificate have been stored by someone with knowledge of the user passphrase, i.e. a client running on one of the user's devices. We chose to use the server to facilitate the cross-device key exchange, because this way the user does not have to transfer key material manually, which is neither secure nor very user-friendly. Most of the client-side encrypted cloud storages we presented earlier, let the users choose their own passwords. Since users rarely choose strong unique passwords, we force them to use a strong randomly generated passphrase. The passphrase consists of English language words. Therefore it is relatively easy to type. The effort required from the user is acceptable, because the user has to provide the passphrase only once for each new device. This effort is repayed by the much better security it provides compared to letting users choose their own passwords as in other solutions such as Tresorit and Sync.

7.3 Revoking a X.509 certificate

Nextcloud does not allow users to change their X.509 certificates and private keys. While it may be true, that users can recover from a lost device, because they have left a note with the passphrase in a secure place or have more devices that still have access to the private key (and passphrase), Nextcloud completely overlooks the threat imposed by a compromised private key (or passphrase). Although the key is stored securely on the device and never transmitted in plaintext, a key compromise may still occur, e.g. if the user loses her device and only has a weak password in place to protect the device's key ring. In such case Nextcloud does not provide any mechanism to revoke access for a lost device and recover from the key compromise, because a user can never change her private key. Nextcloud argues enabling users to renew their certificates and public keys, would open the window for attacks, because an attacker could impersonate the user and provide a new certificate to the server. This threat is similar to the man in the middle attack we will discuss in section 8.1. In Nextcloud, to make such an attack less likely, a user will only once retrieve

another user's X.509 certificate from the server and then store it locally. In all following shares the client will use the same certificate. Because this only gives the server one opportunity to provide a fake certificate, this principle is called Trust On First Use (TOFU).
The ability to revoke a X.509 certificate is considered a fundamental security feature. In our design users do not exclusively base trust in the certificates they retrieve from the server, but have additional mechanisms to verify each other's certificates. Therefore renewing a private/public key pair is acceptable.[49]
Now we will describe, how a user can revoke a compromised private key, e.g. when a device has been lost or compromised.

1. The client, identifying itself with its user credentials, informs the server that the certificate for this username needs to be revoked.

2. The server removes the user's certificate and adds the certificate to the revocation list.

3. If the user still has access to her old private key or the passphrase, she may use it to recover all data from the server and store it locally.

4. The user may then decide to delete the data stored on the server or just remove her own share access and leave the data on the server in case it is shared with another user, that would otherwise lose access.

5. The user generates a new private/public key pair and registers as described in section 7.1. This includes removing the old encrypted private key from the server.

6. The user may decide to re-upload (and re-share) the data she recovered in step 3 and deleted in 4.

Note, that after a revocation the user loses access to all data shared with her. She might have recovered the data in step 3, if she still had access to her old private key, but she can no longer access the shared folders/files in the cloud, because the cryptographic links have been deleted in step 4. Therefore she will have to ask someone with write access to re-share the data with her (using her new certificate). We consider this necessary, because, if the user herself immediately re-shared all data using her new certificate, an attacker, that had compromised the user's account, might do the same. In that case for other users it would be impossible to know, whether an attacker controlled the new certificate or the legitimate user herself. Hence, a manual re-share, including a certificate verification, is the more secure approach.

[49] Tresorit (2017b) even argues, that frequent renewals should take place, because no forward secrecy is provided by the cryptree design and an unacknowledged compromise of a private key, this way would compromise the user's account for all times, theoretically even allowing a potential attacker to gain access to shares that have long been removed from the cloud.

7.4 Sharing data with another user

In this section we describe, how data is shared with another user. The activity diagram in Figure 3 shows how the algorithm works.

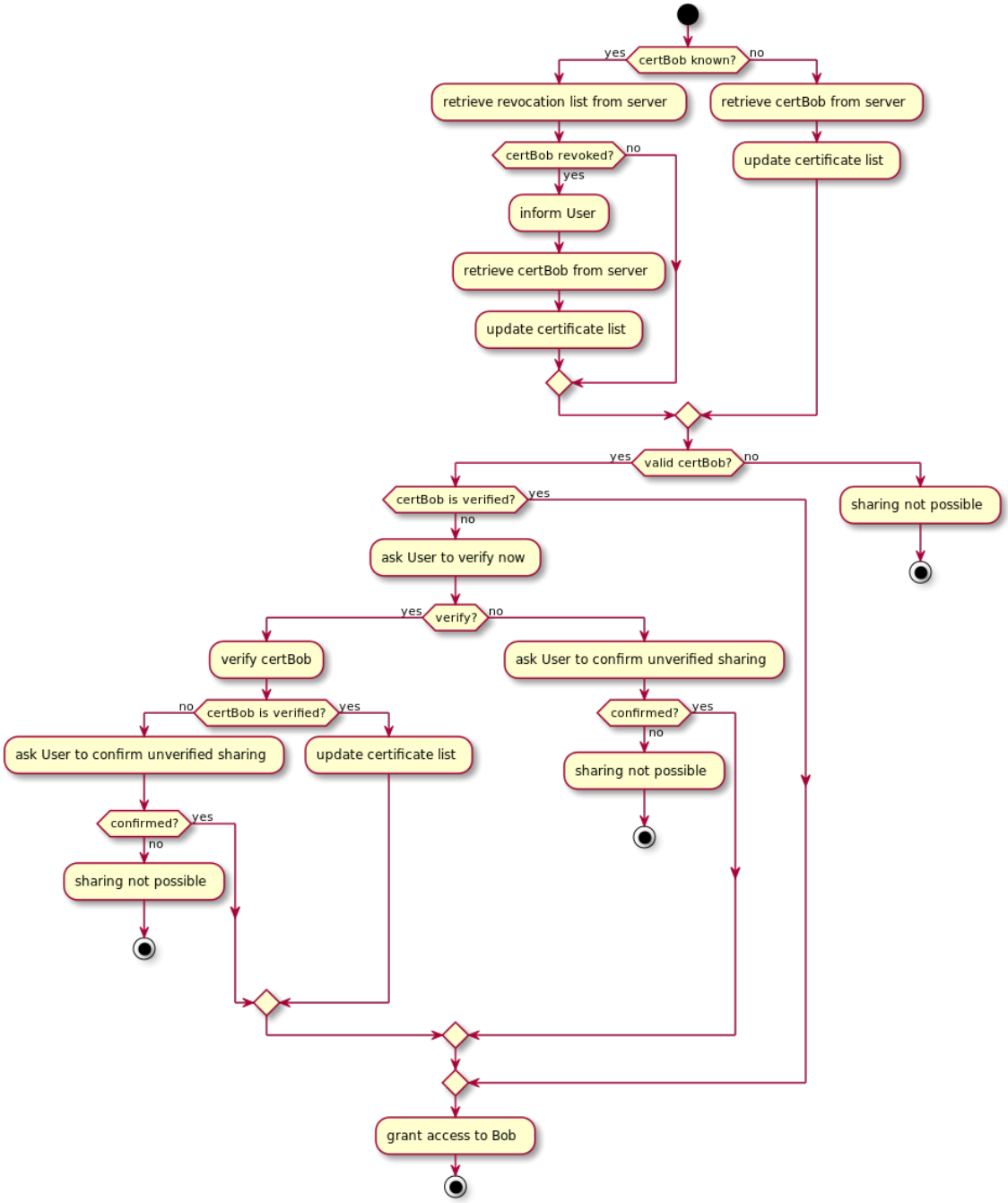

Figure 3: Sharing data activity diagram

1. Check if the user's X.509 certificate is already known.

   (a) If the certificate is not known, fetch it from the server. Verify that it is issued by the CA.

(b) If the certificate is known, fetch the revocation list from the server. Verify the list is issued by the CA.

i. If the certificate has been revoked, try to fetch a new certificate, verify it is issued by the CA and ask the user to confirm the use of the new certificate.

ii. If the certificate has not been revoked, continue with 2.

(c) If no valid certificate is available for the user, inform the user that an encrypted share is currently not possible.

2. Check if the certificate is marked verified.

(a) If the certificate is marked verified, continue with 3.

(b) If the certificate is not marked verified, ask the user to verify it now.

i. If the user does not want to verify the certificate now, ask to confirm the use of an unverified certificate.

ii. Else verify the certificate.

(c) If the user has not confirmed the use of an unverified certificate and no verified certificate is available, inform the user that an encrypted share is currently not possible.

3. Use the certificate to grant access, as described in 6.2.10 and 6.2.11.

(a) If the used certificate, has been updated (i.e. verified and/or replaced by a new one), update the user's (verified) certificate verification list accordingly.

By using this algorithm, we make sure that the user will 1) only use certificates issued by the CA, 2) be informed if a certificate is revoked and a new one is used,[50] 3) be asked to verify other users' certificates and be informed of the problems with using unverified certificates. At first glance the algorithm may seem complicated and requires too much interaction from the user, but the great advantage is, that it gives users the choice to decide for themselves, if a certificate change is valid[51] and if only manually verified certificates should be used or if the user has enough trust in the CA.[52] Note that usually the sequence is that everything happens automatically except the verification of the certificate, which is why the approach is considered an acceptable trade-off between security and usability.
In order to keep track of retrieved and verified certificates a "Certificates" folder and "certificate list" file are created in the "Configuration" directory of the client. The folder is used to store the certificates as retrieved from the server. Each entry in the certificate list contains the following information:

- the username

[50] This is important to recognize possible man in the middle attacks.

[51] The user might want to check with her contact.

[52] Optionally, there might be a (non-default) setting, that allows users to turn off the repeated confirmation of the use of unverified certificates.

- the verification flag (True/False)
- the fingerprint of the X.509 certificate

The user may decide to synchronize this list across devices by using the usual cloud storage. This means it is encrypted with a unique file key and every write operation to the list has to be signed with a key only the user has access to. When the user logs into her account, she may retrieve the list from the server and update it on her device.[53]

8 Public key verification

For the security of public key cryptography it is crucial to ensure the counterparts' public keys are the correct ones. In this section we will first explain how a man in the middle attack on a client-side encrypted cloud storage works. In order to mitigate such attacks, even in case the CSP acts as a man in the middle, we discuss possible solutions and two verification mechanisms for client-side encrypted cloud storages: QR-Code scanning and the Socialist Millionaires' Protocol.

8.1 Why the cloud storage provider should not be trusted as a Certificate Authority

As mentioned before we need to consider the possibility that the server acts as a man in the middle and distributes fake keys in order to get access to user data. Note that it is unlikely, that anybody except the CSP could successfully carry out such an attack, because we use transport encryption and because the CSP is the Certificate Authority. Man in the middle attacks in the context of client-side encrypted cloud storage have been analyzed in detail by Wilson and Ateniese (2014, 406f). The attack works as follows: When Alice wants to share a file with Bob, she asks the CSP for Bob's certificate. The CSP then provides a certificate issued to Bob but containing a public key owned by the CSP. Alice will accept that certificate, because it has been signed by the CA, i.e. the CSP. When she uses the public key to encrypt the file, the CSP can decrypt it. The CSP may then use Bob's real public key to re-encrypt it and send it to Bob. So neither Bob nor Alice will ever know of the attack. This scenario shows how simple it is for the CSP to spoof any identity it wants, because it authorizes *and* distributes public key certificates used for authentication. In their analysis of the CSPs Wuala, SpiderOak and Tresorit Wilson and Ateniese (2014, 406f) found that all three systems acted as a CA and that these systems were therefore vulnerable to this kind of attack.[54] However, the authors could find no evidence any such attack has ever happened. Wilson and Ateniese (2014, 406f) conclude that the CSP cannot be considered a Trusted Third Party and therefore should not act as a CA. Wilson and Ateniese (2014, 411) propose two viable solutions

[53] In our implementation the synchronization has to be performed manually.

[54] They also mention that Tresorit "does not solely rely on the trustworthiness of the certificate authority issuing the users' certificates, but also an invitation secret and a pairing password" (Wilson and Ateniese 2014, 411).

for research: the CSP should 1) allow users to use their own certificates like in a Web of Trust or 2) provide an out-of-band verification mechanism as is the case with off-the-record messaging (OTR). Their findings have been confirmed in a study by Körner et al. (2015, 73), who come to the conclusion that although "the found papers address the issue of data security in general, requirements from the key management have only been expressed rudimentarily."
From their analysis it is clear that we can use the server for key distribution, but cannot solely rely on it for key verification. The proposed Web of Trust requires too much interaction from the users and has too many pitfalls as described in section 3.4.1. Hence we us a PKI and provide secure mechanisms for certificate verification to make man in the middle attacks by the CSP infeasible.

8.2 QR-Codes scanning

In recent years we have seen a dramatic increase in the use of mobile devices. Körner and Walter (2014, 224) point out, that the use of mobile devices does not only bring new challenges, but also allows for new forms of interactions, we might leverage to facilitate key exchange and key verification. In their paper they propose QR-Codes displayed on one user's device and scanned by another device as a viable mechanism for key exchange and key verification.[55] We have already resolved the problem of key exchange and our design has the advantage that it on the one hand allows users to recover the private key after device loss and on the other hand allows adding devices without a camera. Therefore in this section we will focus on using QR-Codes for key verification.
First note that QR-Codes count with the necessary capacities to transfer asymmetric keys and of course their much shorter fingerprints. The largest QR-Codes allow to encode 2953 bytes (15624 bits) (Körner and Walter 2014, 225). Considering that the largest RSA keys in use today are 4096 bits long, this is enough capacity. Note that QR-Codes contain checksums that serve to detect errors in the transfer.
QR-Codes can also be used for key verification, which is already implemented in mobile messengers like Signal or Threema. If Alice and Bob want to verify each other's key, one of them, say Alice, generates a QR-Code. For this purpose she computes the SHA256 hash value of the concatenation of the fingerprints of both her and Bob's certificate. She then encodes this value in her QR-Code. Bob performs the same computation using the certificates on his device, then he scans Alice's QR-Code and compares the values generated by Alice and by himself. If they match, Alice and Bob can mark each other's certificates as verified. Of course Alice and Bob can always read out the fingerprints to one another, but it is much more convenient to do it with a simple QR-Code scan. The advantage of using this method is, that one scan is enough to make the verification mutual, because both certificates are used for the computation.

[55] QR-Codes can be used for key exchange as follows: If Alice wants to move her private/public key pair to a new device, her client generates a QR-Code that contains both keys and displays it on the device. The client on the new device scans it, retrieves both keys and after checking the X.509 certificate stores them securely on the new device. If Alice is in an insecure environment she might even encrypt the keys with a password using a function such as PBKDF2.

While QR-Codes are a powerful and secure tool on mobile devices that have a camera, they are not very useful for desktop computers and other devices without a camera. It will be impossible to use QR-Codes to verify certificates on such devices. Also QR-Code scanning requires Alice and Bob to meet physically and bring their mobile devices in order to verify each other's keys. We assume such meeting will happen often enough to make QR-Codes a handy instrument, but for the case that it physical meeting is not possible, we provide another verification mechanism, described in the following section.

8.3 Socialist Millionaires' Protocol

In this section we will explain how the Socialist Millionaires' Protocol (SMP) can be used for key verification as an alternative to QR-Codes. The main benefit of the SMP is, that it does not require the communicating parties to use an out-of-band channel or meet physically at the time they verify each other's public keys.[56] However, since the SMP has originally been designed for instant messaging, it requires synchronous communication in order to be secure. Although it would technically be possible to have the CSP exchange the messages asynchronously, that would slow down the process intolerably, because the SMP requires several message exchanges. Furthermore this would make it much easier for an attacker to brute force the secret, because the attacker would have much more time for the attack. Therefore, it is essential that the protocol is performed in a synchronous session. In a client-side encrypted cloud storage system the original protocol as explained in section 3.4.4 needs to be modified as follows in order to provide key verification.
Alice sends a request to the server asking to verify Bob's key via the SMP. In this request she includes a question, which only Bob knows the answer to. Alice also provides the expected answer to the question, but this answer will never be transmitted. The server informs Bob of Alice's request. If Bob is online and accepts it, before the timeout expires, the actual protocol is performed. That means Bob will answer the question, providing the shared secret and after that Alice's and Bob's client application exchange the necessary messages relaying them through the server. If the protocol has terminated successfully within the time limit of one minute, both Alice and Bob mark their certificates as verified and are informed of the outcome of the verification. Note, that we have arbitrarily set the time limit to one minute, because it is assumed to be infeasible to brute force the secret in such short time. However, a proper evaluation of what would be an adequate time limit should be performed, before this system goes in production.

9 Implementation of the proof of concept

In this section we will outline the details of our proof of concept implementation. It consists of a server application that is designed to act as a middleware on top of an existing potentially insecure cloud storage, that provides reliability and availability for the stored data. Both the client and server are written in Python 3.5. For

[56] In any key verification it is necessary to exchange information over an out-of-band channel (Körner and Walter 2014, 223f). In SMP this information is represented in the shared secret.

cryptography we use the Python module cryptography in both client and server application, with the exception of the SMP, which is our own implementation, because there are no audited libraries available in Python.
It is important to point out that our applications both client and server are merely a proof of concept. We did not intend to apply secure programming at all. At many points in our code exceptions and corner cases are not handled appropriately, because we only tested it manually and did not write any unit tests etc. Therefore our application should not be used in a production environment. Another reason is, that both applications currently do not store private keys securely in the Operating System's key ring (or by setting appropriate file access rights) but simply store keys as a file in the application folder. For more details about these limitations have a look at our TU-Berlin GitLab Repository, where we also published our code.[57]

9.1 Server

The main purpose of the server application is on the one hand to act as a CA and the key distributor in a PKI and on the other hand store the users' confidential data. The server is a small Flask application of about 1000 lines of code. It provides a RESTful API which the client uses to retrieve and modify data stored by the server. The server has access to a Mysql database, which stores all information on user accounts, certificates and revocations for the PKI and even the data, stored by the user. The database and especially user account data is also used to manage access control.
We do not use a real cloud storage in our proof of concept, but only store the users' data in a database.[58] This is acceptable, because our focus is mostly on the client, where the encryption/decryption operations happen, and it is not at all on proving the usage and security guarantees that cloud storages provide. The server application also stores notifications, which clients can retrieve in order to be informed of new shared data with the user or new SMP certificate verification requests.

9.2 Client

The client application consists of about 1700 lines of code. Currently we only implemented a Desktop client, but there is no reason not to port our code to offer mobile clients. Our application uses Python's requests module for connection's with the server. After login (and registration) it shows a main menu, in which the user can choose between several options regarding file management and certificate verification. It also constantly requests notifications from the server, in order to inform the user of new data accessible or new SMP verification requests.
The client does not implement the complete cryptographic read and write trees as proposed in 6. Since similar cloud storages are already working in practice, in our proof of concept we focus on the key verification and key exchange mechanisms instead. That means sharing and uploading files to the cloud in our implementation is only a mockup we implemented to prove that our PKI, key exchange and key

[57] `https://gitlab.tubit.tu-berlin.de/gaspar_ilom/crypt-cloud`
[58] For this proof of concept implementation we can store files up to 4GB of size (after encryption).

verification mechanisms really enable users to share data securely. The user is only able to upload individual files and has to do so manually. Before uploading files the file data and their names are encrypted with AES-128-CBC using a PKCS#7 padding and HMAC-SHA256 for authentication and integrity. Because both AES and HMAC are based on the same key, there is no separation of write and read access in this proof of concept. The 128 bit AES key is encrypted with the user's private key and also uploaded to the server. For sharing it can be re-encrypted with another user's private key and attached to the original key data on the server. This is the equivalent of what we earlier referred to as a cryptographic link. In our implementation it is possible to revoke a share by removing the encrypted key for a user from the server,[59] but we did not implement lazy re-encryption as proposed in section 6, so the actual key used for encryption will not be changed.[60] The revocation of a share does not require access to the user's private key or a revocation certificate, but only to the the user's login credentials (username/email and password). In production this should be changed, because otherwise an attacker might revoke a user's certificate, in order to forge a new one controlled by her.[61]
When adding the initial device, the client registers an account and then generates a private key as explained in section 7.1. The private key is encrypted with the random generated 12 words passphrase, according to the PKCS#8 standard. This means PBKDF2 with 2048 iterations of HMAC-SHA256 and a 64 bit random salt is used to derive a key and use AES-256-CBC to encrypt the private key with the derived key. We point out that we use cryptography's BestAvailableEncryption class, so that this algorithm might change over time. After encrypting it the private key is stored both on the server and the user's device. The client also requests a corresponding X.509 certificate from the server. The twelve word passphrase is displayed to the user (to remember it or write it down and store it in a safe place). It is also stored on the device. If the user ever loses her device or wants to add a new one, she can retrieve the private key from the server and decrypt it with the twelve word passphrase as explained in section 7.2.
Once the the client is configured, i.e. the user registered, logged in and has a valid certificate and corresponding private key, the user can upload data to the cloud and share it. For sharing we implemented our algorithm as explained in section 7.4. In this algorithm the user is asked to verify the counterparts certificate. For doing so, we provide two options: QR-Codes and the Socialist Millionaires' Protocol.
The activity diagram in Figure 4 shows how QR-Code verification works. The client uses the fingerprints of both users' certificates in order to compute a hash value. If the user wants to display the QR-Code this value is Base64 encoded before generating and displaying the QR-Code. The Base64 encoding is necessary, because the Python library used to scan the QR-Codes, zbar-py, has some issues when scanning non-ASCII characters. Since the SHA256 hash value is only 32 bytes long this is not a problem. After displaying the QR-Code the user is asked, whether

[59] The server also implements an ACL, in order to control who has access to a file.

[60] The only way to change the key currently is to re-upload the file (and re-share it, if necessary).

[61] On the other hand requiring a signature from the private key or a revocation certificate means that some users will not be able to revoke their private keys, which also represents a security threat. So in this case it should really be ensured that users store their revocation certificates in a safe place, they will not lose access to.

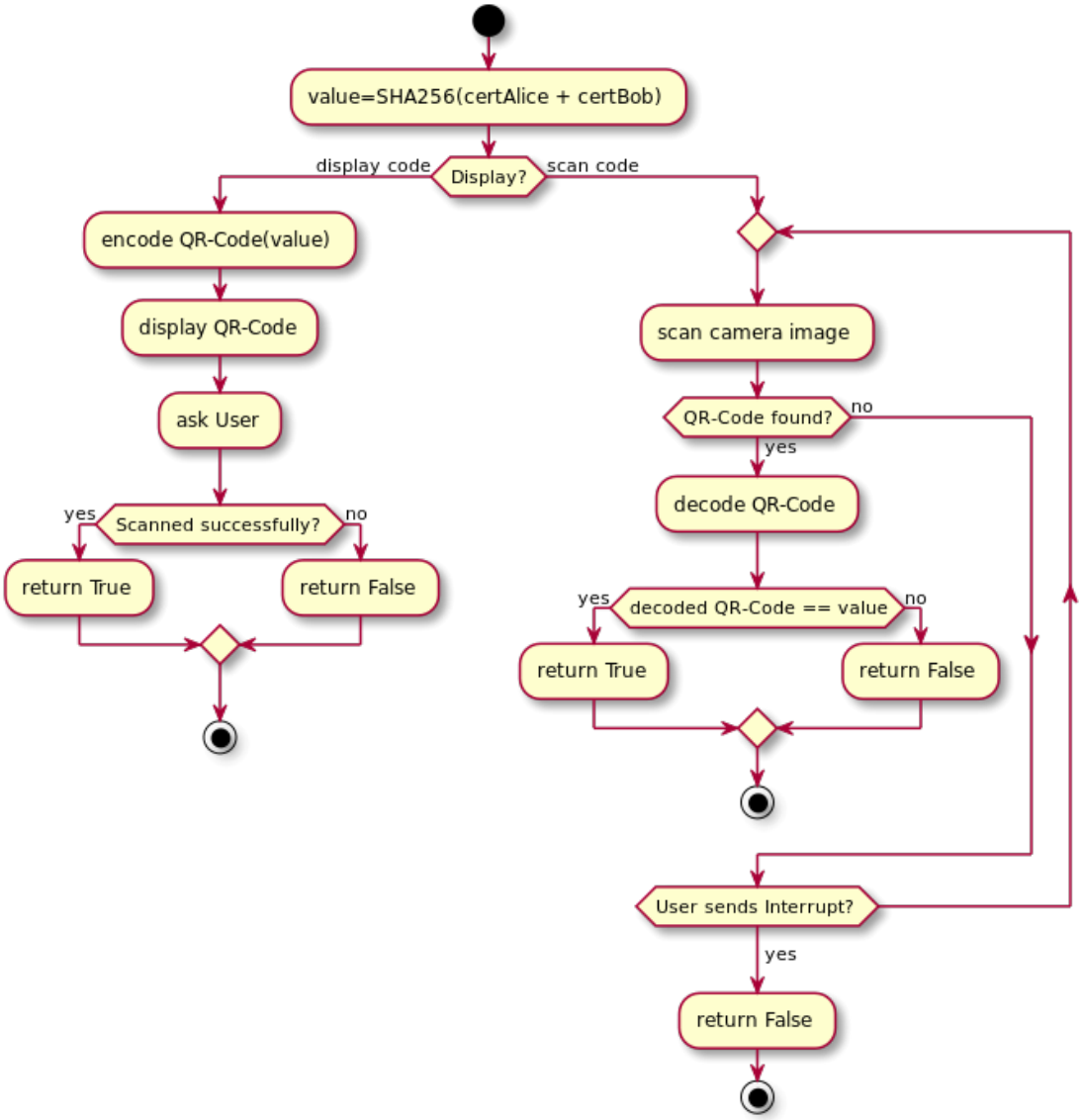

Figure 4: QR-Code verification activity diagram
Note that *value* actually is calculated from the SHA256 fingerprints of both certificates. QR-Codes are Base64 encoded.

the scanning on her counterpart's device succeeded. If so, she can answer yes and the counterparts certificate is marked as verified. This is possible, because we use both users' certificates to compute the fingerprint and therefore the verification is always mutual.[62] If the user scans the QR-Code, the client will read images from the camera of the device and try to find a QR-Code. Once it has found a QR-Code, it decodes the read value and compares it to the hash value computed earlier. If the

[62]Theoretically it is possible that the counterpart uses a manipulated version of the client application, that will always display a successful result. That way the user could be tricked into verifying the counterpart's certificate. However, if the counterpart's client really was manipulated by an attacker, the attacker would probably have access to all files and the user's private key anyway. So this does not represent a weakness an attacker could exploit to gain access to new data.

values match, the user can be certain, that the same certificates are used on both devices.

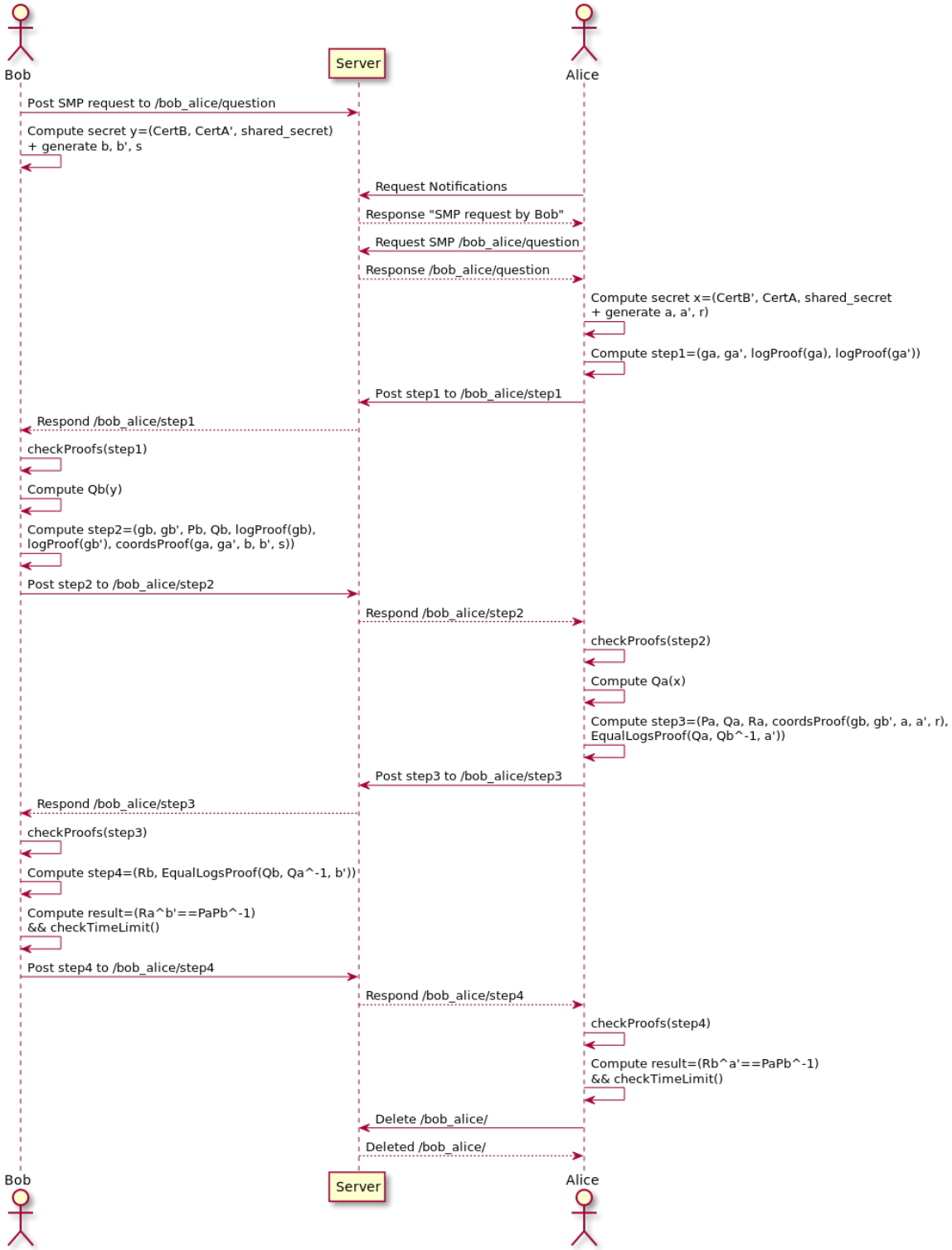

Figure 5: SMP verification sequence diagram
Note that not all requests by the clients to the server are included, because the diagram is already complex enough.

In order to implement the Socialist Millionaires' Protocol we modified Shane Tully's (2013) version of the SMP according to our needs. Most importantly we converted his code to Python3 and modified the secret generation. In our code the secret is a SHA256 hash value computed from a concatenation of the SHA256 fingerprints of both participants' certificates and the shared secret. We point out that the shared secret and thus the computed secret are never transmitted over a network. Our clients do not open direct connections to one another, but relay their messages by posting them on the server. The complete interaction, as we implement it, is displayed in Figure 5. Each of the messages posted to the API provided by the server is Base64 encoded, in order to be more easily handled by the flask-RESTful module.[63]
We considered opening a peer to peer connection between the clients in order to perform the SMP verification, but finally decided against that. It is not necessary, because the SMP was created exactly for a scenario in which the server might act as a man in the middle.[64] The protocol ensures, that a malicious server will be detected. The SMP requires a synchronous session and we arbitrarily set a time limit to one minute. If the protocol has not terminated one minute after the client posted the first cryptographically relevant data, it is aborted and the verification fails. Further research is required in order to figure out, what would be a time limit that on the one hand makes sure a man in the middle does not have enough time to brute force a weak natural language secret and on the other hand still provide enough time for the client to successfully finish the protocol. It is important to point out that neither our implementation nor Shane Tully's have been security audited. So it should not be used if the confidentiality of the shared data is critical.

10 Discussion

In this section we discuss our design and the proof of concept. Note that we already explained some differences between our approach and the related work in section 4. We will not repeat all differences in detail here. Our goal was to engineer a cloud storage system, that allows users to share data with client-side encryption in place. We achieved that with our proof of concept implementation. In section 6 we even proposed a cryptographic key structure, that is more efficient and powerful than what we implemented. Other evaluations, e.g. by Grolimund et al. (2006) have shown the efficiency of similar cryptographic tree structures, which is why we did not include it in our proof of concept. Due to the cryptographic tree structure the use of computing power and bandwidth is limited to a minimum in our design. For instance, granting access does not require any re-encryption operations and only one cryptographic link is added per user, because users inherit all downward access rights from the shared folder.

[63]The computational and spatial overhead is not considered relevant for a proof of concept application, but the Base64 encoding could be avoided if required to enhance performance (see section 10).

[64]It is used in a similar context in OTR, where an XMPP server might act as a man in the middle.

We implemented a mechanism that allows users to exchange their private key between devices by storing it encrypted on the server and retrieve it from there. All they need to remember (or store in a save place) is a 12 English words passphrase. Alternatively users can display it on one of the devices, they already added to the account. The advantage of our design is that 12 randomly chosen words provide enough entropy not to be easily brute forced by an attacker, which would be more likely, if we let users choose their own passwords. Of all existing client-side encrypted CSPs we evaluated in section 4.2 Nextcloud is the only one, that enforces strong passwords to protect the users' private keys. Contrary to Nextcloud, that implements a similar key exchange mechanism, our approach allows certificate revocations in order to minimize the threat a private key compromise represents. Although this design achieves a high level of security, further research is needed regarding the usability of the key exchange mechanism, i.e. whether users actually find it acceptable to use such a long passphrase or whether better solutions to enforce security are needed.

We developed a small PKI, in which the server application acts as the CA and key distributor. Since PKIs are well-proven concepts, we will not discuss it here. Once a user has retrieved another user's certificate from the PKI, she can verify it either using QR-Code verification or SMP verification. None of the CSPs we evaluated provides proper certificate verification, which is why our system is the only one that does not require users to trust the CSP.[65] Both QR-Codes and the SMP provide technically secure mutual certificate verification. QR-Codes usually require users to meet physically, although they might also be shared over an out-of-band channel, so that no physical meeting is necessary. QR-Code verification therefore is very similar to manually comparing the fingerprints of the certificates, but it is much more convenient.

The Socialist Millionaires' Protocol in contrast neither requires a physical meeting nor an out-of-band channel for certificate verification, but it requires a synchronous session in order to be secure. Otherwise we could not use a weak natural language secret, because an attacker might have enough time to brute force the secret. The main problem with the SMP is, that it remains uncertain, whether users actually trust the protocol (which is too complicated to be understood by the average user) and find it acceptable to require a synchronous session to verify their certificates. SMP and QR-Code verification are supposed to complement each other according to the users' needs and preferences. Practical experience has to show, whether that works as intended.

Even if users do not verify other users' certificates they are still able to share data. Because this means they have full trust in the CSP as a CA, we implemented our sharing logic such, that it strongly encourages users to perform a certificate verification. Again this might be annoying for some users, but hopefully not for high profile users (like activists and journalists) that are aware of the threat an insecure cloud storage represents.

Summarizing our discussion we state, that we achieved to design a system that outreaches the degree of security of existing client-side encrypted cloud storages.

[65] Tresorit implements a verification mechanism, which is not openly documented and which uses a secret generated by their servers. Therefore we do not consider it trustworthy.

We achieved this without reinventing the wheel, but rather by optimizing existing solutions (e.g. the key exchange from Nextcloud), introducing algorithms from other fields (e.g. the SMP from instant messaging) and combining these pieces in a new fertile way. Unfortunately, as mentioned above we were not able to achieve all of our security goals without making some concessions regarding usability. Nevertheless, we believe our design is a viable option for users, that appreciate being able to control who gets access to their data without requiring a trusted third party.
Our concepts have proven to work technically. We decided not to run benchmarks and evaluate the performance, because we believe that performance is not an issue for the system we designed due to the following reasons. The innovative parts of our system mostly concern the mechanism to exchange keys between devices and verification of other users' certificates. QR-Code verification does not require any interaction with the server and only two users will verify each others keys at a time by transmitting data exclusively within the QR-Code. No data is stored permanently on either device. Therefore QR-Code scanning does not represent a performance issue. In regard to SMP verification the server has to store a natural language question and four messages which range from 428 to 1676 bytes. In our implementation the messages are Base64 encoded before sending, which means 3 bytes will be turned into 4 bytes effectively in transmission and storage at the server. So the maximum message size is 2235 bytes to be transmitted and stored on the server. Considering that the server should be able to retrieve and deliver large files of hundreds of MBs and that the SMP data will only be stored for less than a minute, there is no problem regarding performance. Note, that the server does not have to perform any calculations and the calculations performed by the clients are comparable to several Diffie Hellman Key Exchanges, which is acceptable, because the protocol will only occasionally be performed.
The user's private keys and certificates (one of each per user) are just a few thousand bytes large and it is unproblematic to store them in the database. They will only be accessed when registering a new account, adding a new device to that account or sharing data with a new contact. So it should happen much less frequently than the usual data access. Therefore we conclude that our key exchange and verification mechanisms will not be the bottleneck of the cloud storage neither at client- nor at server-side.

11 Conclusion

This thesis presented a new client-side encrypted cloud storage system and successfully proved its feasibility with a proof of concept implementation. We introduced an improved version of the cryptographic tree structure Cryptree, which enables us to flexibly and intuitively share data and distinguish between read and write access. A unique encryption key is used for every file, which reduces the amount of necessary encryption/decryption operations to a minimum, and thus provides the basis for an efficient client-side encrypted cloud storage. Future research could go even further and find ways of using functional cryptography in order to avoid the re-encryption and re-upload of a complete file after each modification while also providing file versioning and simultaneous modification of encrypted documents by

multiple users.
We presented a cross-device key exchange mechanism that allows users to flexibly add (and remove) devices to their account and even recover from the loss of all devices. The innovation is that this is possible without a security trade-off. Because the user's private key is encrypted with a random generated 12 words passphrase and stored in the cloud, it is ensured that the encryption key used to protect the private key has enough entropy and still is easily recoverable for the user. This does not only protect the confidentiality of the data in the cloud, but also its availability, because users will not easily lose access to their accounts. Contrary to existing solution our design does not require any trust in the server, because it provides efficient and easy to use means for certificate verification. This is achieved by either QR-Code scanning or the Socialist Millionaires' Protocol. While the former requires users to physically meet or send the QR-Code over an out-of-band channel to mutually verify their certificates, the latter can be performed over an insecure channel. This insecure channel may even be controlled by the CSP itself, as it is the case in our implementation. All that is required from the users is a synchronous session and a shared secret, that can even be a dictionary word.
The client-side encrypted cloud storage system we designed is usable and secure. It preserves the complete Sync and Share functionality with the exception of web access, which is not included, because there is no way to provide it without implicitly requiring users to trust the code delivered by the untrusted CSP and executed in the browser for cryptographic operations. Our design will be published under an open source license, so that anyone willing is able to verify the client application really works as we claim. Due to the newly introduced features we were able to present a new client-side encrypted cloud storage system that exceeds the degree of security guaranteed by former solutions. Therefore our system is particularly suitable for users, that must heavily rely on the confidentiality, integrity and authenticity of their data such as activists, lawyers, journalists and others.

References

Amazon. 2018. *Amazon S3 Reduced Redundancy Storage*. Visited on 03/16/2018. https://aws.amazon.com/s3/reduced-redundancy/.

Bourrier, Mathieu. 2016. *ownCrypt_ CryptoDesign-1*. Visited on 06/23/2017. https://cloud.airmail.fr/s/d475f3c9d77e99d8b413977729be5f0e/download.

– . 2015. *ownCrypt_ Intents&ThreatModel&Features-2*. Visited on 06/23/2017. https://cloud.airmail.fr/s/a9fe1a55a59374dd6bc399f4ff4d60db/download.

Cachin, Christian, Idit Keidar, and Alexander Shraer. 2009. "Trusting the Cloud". *SIGACT News* 40 (2): 81–86.

Cryptomator. 2017a. *Cryptomator - Homepage*. Visited on 09/21/2017. https://cryptomator.org.

– . 2017b. *Cryptomator - Sicherheitsarchitektur*. Visited on 09/21/2017. https://cryptomator.org/architecture/.

Delfs, Hans, and Helmut Knebl. 2015. *Introduction to Cryptography*. Information Security and Cryptography. DOI: 10.1007/978-3-662-47974-2. Berlin, Heidelberg: Springer Berlin Heidelberg.

Diffie, W. 1992. "The First Ten Years of Public-Key Cryptology". In *Contemporary Cryptology: The Science of Information Integrity*, 135–175. IEEE Press New York.

Diffie, W., and M. Hellman. 1976. "New directions in cryptography". *IEEE Transactions on Information Theory* 22 (6): 644–654.

Ertel, Wolfgang. 2012. *Angewandte Kryptographie*. München: Hanser.

Ferguson, Niels. 2010. *Cryptography engineering : design principles and practical applications*. Indianapolis, Ind.: Wiley.

Frosch, T., C. Mainka, C. Bader, F. Bergsma, J. Schwenk, and T. Holz. 2016. "How Secure is TextSecure?" In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, 457–472.

Gadhavi, L., M. Bhavsar, M. Bhatnagar, and S. Vasoya. 2016. "Design of efficient algorithm for secured key exchange over Cloud Computing". In *2016 6th International Conference - Cloud System and Big Data Engineering (Confluence)*, 180–187.

Grolimund, D., L. Meisser, S. Schmid, and R. Wattenhofer. 2006. "Cryptree: A Folder Tree Structure for Cryptographic File Systems". In *2006 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, 189–198.

Hallberg, Sven Moritz. 2008. "Individuelle Schlüsselverifikation via Socialist Millionaires' Protocol". Visited on 09/23/2017. http://www.khjk.org/log/2012/apr/smp/smp.pdf.

IETF. 2008. *RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. Visited on 10/11/2017. https://tools.ietf.org/html/rfc5280.

Kamara, Seny, and Kristin Lauter. 2010. “Cryptographic Cloud Storage”. In *Financial Cryptography and Data Security*, 136–149. Springer, Heidelberg.

Kaur, Manpreet, and Rajbir Singh. 2013. “Implementing encryption algorithms to enhance data security of cloud in cloud computing”. *International Journal of Computer Applications* 70 (18).

Kim, Tiffany Hyun-Jin, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. 2013. “Accountable Key Infrastructure (AKI): A Proposal for a Public-key Validation Infrastructure”. In *Proceedings of the 22Nd International Conference on World Wide Web*, 679–690. WWW ’13. Rio de Janeiro, Brazil: ACM.

Kiss, Jemima. 2014. “Snowden: Dropbox is hostile to privacy, unlike ’zero knowledge’ Spideroak”. *The Guardian* (London). Visited on 09/21/2017. `http://www.theguardian.com/technology/2014/jul/17/edward-snowden-dropbox-privacy-spideroak`.

Körner, Kevin, Holger Kühner, Julia Neudecker, Hannes Hartenstein, and Thomas Walter. 2015. “Bewertungskriterien und ihre Anwendung zur Evaluation und Entwicklung sicherer Sync&Share-Dienste.” In *DFN-Forum Kommunikationstechnologien*, 71–82. Visited on 09/27/2017. `http://cs.emis.de/LNI/Proceedings/Proceedings243/71.pdf`.

Körner, Kevin, and Thomas Walter. 2014. “Sichere und benutzerfreundliche Schlüsselverteilung auf Basis von QR-Codes.” In *GI-Jahrestagung*, 223–234.

Kumar, A., B. G. Lee, H. Lee, and A. Kumari. 2012. “Secure storage and access of data in cloud computing”. In *2012 International Conference on ICT Convergence (ICTC)*, 336–339.

Lám, I., S. Szebeni, and L. Buttyán. 2012a. “Invitation-Oriented TGDH: Key Management for Dynamic Groups in an Asynchronous Communication Model”. In *2012 41st International Conference on Parallel Processing Workshops*, 269–276.

– . 2012b. “Tresorium: Cryptographic File System for Dynamic Groups over Untrusted Cloud Storage”. In *2012 41st International Conference on Parallel Processing Workshops*, 296–303.

Lám, István. 2016. *Was ist Zero-Knowledge-Verschlüsselung?* Visited on 09/17/2017. `https://tresorit.com/blog/zero-knowledge-verschlusselung/`.

Law, Effie Lai-Chong, Virpi Roto, Marc Hassenzahl, Arnold P.O.S. Vermeeren, and Joke Kort. 2009. “Understanding, Scoping and Defining User Experience: A Survey Approach”. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 719–728. CHI ’09. Boston, MA, USA: ACM.

Linux Mint. 2016. *Beware of hacked ISOs if you downloaded Linux Mint on February 20th! – The Linux Mint Blog*. Visited on 10/16/2017. `https://blog.linuxmint.com/?p=2994`.

Meister, Andre. 2017. “Staatstrojaner: Bundestag hat das krasseste Überwachungsgesetz der Legislaturperiode beschlossen (Updates)”. *netzpolitik.org*. Visited on 10/21/2017. https://netzpolitik.org/2017/staatstrojaner-bundestag-beschliesst-diese-woche-das-krasseste-ueberwachungsgesetz-der-legislaturperiode/.

Melara, Marcela S., Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. 2015. “CONIKS: Bringing Key Transparency to End Users”. In *24th USENIX Security Symposium (USENIX Security 15)*, 383–398. Washington, D.C.: USENIX Association.

Nextcloud. 2017a. *End-to-end encryption design*. Visited on 10/01/2017. https://nextcloud.com/endtoend/.

– . 2017b. *Nextcloud*. Visited on 10/01/2017. https://nextcloud.com/.

Nielsen, Jakob. 2017. *Usability 101: Introduction to Usability*. https://www.nngroup.com/articles/usability-101-introduction-to-usability/.

OTR Development Team. 2017. *Off-the-Record Messaging Protocol version 3*. Visited on 10/12/2017. https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html.

Pancholi, Vishal R, and Bhadresh P Patel. 2016. “Enhancement of cloud computing security with secure data storage using AES”. *International Journal for Innovative Research in Science and Technology* 2 (9): 18–21.

Percival, Colin. 2009. “Stronger key derivation via sequential memory-hard functions”, 1–16. BSD Can. Visited on 09/27/2017. http://www.bsdcan.org/2009/schedule/attachments/87_scrypt.pdf.

PrivateBin. 2016. *WebExtension: A proof-of-concept webextension for verifying JS integrity*. Original-date: 2016-12-11T18:36:50Z. Visited on 10/16/2017. https://github.com/PrivateBin/WebExtension.

Raymond. 2015. *SafeMonk: Dropbox + Security Made Simple*. Visited on 09/22/2017. http://www.dropboxwiki.com/dropbox-addons/safemonk-dropbox-security-made-simple.

Rewagad, P., and Y. Pawar. 2013. “Use of Digital Signature with Diffie Hellman Key Exchange and AES Encryption Algorithm to Enhance Data Security in Cloud Computing”. In *2013 International Conference on Communication Systems and Network Technologies*, 437–439.

Rivest, R. L., A. Shamir, and L. Adleman. 1978. “A method for obtaining digital signatures and public-key cryptosystems”. *Communications of the ACM* 21 (2): 120–126. Visited on 09/15/2017. http://portal.acm.org/citation.cfm?doid=359340.359342.

Rueppel, Rainer A. 1992. “Stream Ciphers”. In *Contemporary Cryptology: The Science of Information Integrity*, ed. by G.J. Simmons, 65–134. IEEE Press New York.

SafeMonk. 2013. *Key Management and Organization*. Visited on 08/13/2013. http://support.safemonk.com/customer/portal/articles/1018901-key-management-and-organization.

Scherschel, Fabian A. 2015. "Adieu sichere Dropbox: Cloud-Verschlüsseler SafeMonk gibt auf". *heise Security*. Visited on 09/22/2017. http://www.heise.de/security/meldung/Adieu-sichere-Dropbox-Cloud-Verschluesseler-SafeMonk-gibt-auf-2637330.html.

Schneier, Bruce. 1996. *Applied Cryptography : Protocols, Algorithms, and Source Code in C*. 2. ed. New York: Wiley.

Seafile. 2017. *Security features · Seafile Server Manual*. Visited on 10/01/2017. https://manual.seafile.com/security/security_features.html.

Sookasa. 2017. *SOOKASA WHITEPAPER | SECURITY*. Visited on 09/22/2017. https://www.sookasa.com/wp-content/uploads/2016/01/Jan2016_Sookasa_WhitePaper_Security.pdf.

SpiderOak. 2017a. *Encryption White Paper*. Visited on 09/21/2017. https://spideroak.com/resources/encryption-white-paper.

– . 2017b. *No Knowledge, Secure-by-Default Products*. Visited on 09/17/2017. https://spideroak.com/no-knowledge/.

– . 2017c. *Why We Will No Longer Use the Phrase Zero Knowledge to Describe Our Software*. Visited on 09/17/2017. https://spideroak.com/articles/why-we-will-no-longer-use-the-phrase-zero-knowledge-to-describe-our-software/.

Swoboda, Joachim, Stephan Spitz, and Michael Pramateftakis. 2008. *Kryptographie und IT-Sicherheit : Grundlagen und Anwendungen*. 1. Aufl. Studium : IT-Sicherheit und Datenschutz. Wiesbaden: Vieweg + Teubner.

Sync. 2017a. *Privacy Matters*. Visited on 09/26/2017. https://www.sync.com/your-privacy/.

– . 2017b. *Privacy White Paper*. Visited on 09/26/2017. https://www.sync.com/pdf/sync-privacy.pdf.

Syncplicity. 2015. *Syncplicity Security and Control Features*. Visited on 09/21/2017. https://www.syncplicity.com/resources/datasheets/syncplicity-security-and-control-features.

T., Finanzmagazin I. 2014. *Zero Knowledge wird zum Datentransfer-Gütesiegel · IT Finanzmagazin*. Visited on 09/21/2017. https://www.it-finanzmagazin.de/zero-knowledge-wird-zum-datentransfer-guetesiegel-1888/.

Tresorit. 2017a. *Tresorit - Security*. Visited on 09/26/2017. https://tresorit.com/security.

– . 2017b. *White Paper*. Visited on 09/26/2017. https://tresorit.com/files/tresoritwhitepaper.pdf.

Tully, Shane. 2013. *MITM Protection via the Socialist Millionaire Protocol (OTR-style)*. Blog. Visited on 03/13/2018. https://shanetully.com/2013/08/mitm-protection-via-the-socialist-millionaire-protocol-otr-style/.

W3C. 2016. *Subresource Integrity*. Visited on 10/16/2017. https://www.w3.org/TR/SRI/.

Whitten, Alma, and J. D. Tygar. 1998. *Usability of Security: A Case Study,* tech. rep. CMU-CS-98-155. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE. Visited on 09/29/2017. `http://www.dtic.mil/docs/citations/ADA361032`.

Wilson, Duane C., and Giuseppe Ateniese. 2014. “"To Share or not to Share" in Client-Side Encrypted Clouds”. In *Information Security: 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings*, ed. by Sherman S. M. Chow, Jan Camenisch, Lucas C. K. Hui, and Siu Ming Yiu, 401–412. Cham: Springer International Publishing.

Yao, F.F., and Y.L. Yin. 2005. “Design and Analysis of Password-Based Key Derivation Functions”. *IEEE Transactions on Information Theory* 51 (9): 3292–3297. Visited on 09/14/2017. `http://ieeexplore.ieee.org/document/1499059/`.

Zhao, G., C. Rong, J. Li, F. Zhang, and Y. Tang. 2010. “Trusted Data Sharing over Untrusted Cloud Storage Providers”. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, 97–103.

Zwattendorfer, Bernd, Bojan Suzic, Peter Teufl, and Andreas Derler. 2013. “Sicheres Speichern in der Public Cloud mittels Smart Cards”. In *D-A-CH Security*, 120–132. Nürnberg.