# Problem Description

Modern cloud storage systems that encrypts data for "at rest" protection often has access to the stored data, since the encryption is performed with an encryption key known by the service provider.

The candidates will design an application in which all data stored in the cloud are encrypted with a key that can not be obtained by the service provider.

Other goals of the application should be the ability to share encrypted files with others users, without leaking the encryption key to the service provider, and make it as user friendly as possible without compromising security. The students will develop a proof of concept implementation of the design.

| | |
|---|---|
| Assignment given: | 24. January 2011 |
| Supervisor: | Danilo Gligoroski |
| External Supervisors: | Carsten Maartmann-Moe, Ernst & Young AS |
| | Antonio Martiradonna, Ernst & Young AS |

# Abstract

Today, major IT-companies, such as Microsoft, Amazon and Google, are offering online storage services to their customers. This is a favourable solution – as opposed to regular storage – in terms of low costs, reliability, scalability and capacity, however important security features such as data privacy and integrity are absent.

To address these issues, a cryptographic architecture is proposed, that ensures the confidentiality and integrity of the data stored by users, independent of the trust of the underlying provider. It also includes secure sharing of private data among customers of the same provider. The scheme was further implemented in a proof of concept server and client.

The underlying cryptographic architecture is based on existing open source systems and cryptographic primitives. The architecture was further implemented as a highly reusable general library in Java. An Android client was created and several performance tests were conducted.

The proof of concept system shows that it is possible to implement the proposed scheme and that the cryptographic operations does not significantly affect the user experience on an Android device. Weaknesses of the scheme are identified, and the key that has to be stored client side is found to be the greatest risk.

We have presented a scheme for secure storage and sharing of files on an untrusted server, and argued for its validity. To support streaming functionality, the scheme could be extended with hash trees to validate small parts of a file at the time.

# Preface

The work behind this report was carried out during the spring semester in 2011 at the Norwegian University of Science and Technology (NTNU), Institute of Telematics (ITEM). The report is the final result of a master thesis in information security written by three graduate students at NTNU. The thesis was assigned by Ernst & Young AS Norway, and acknowledged by ITEM.

We would like to thank our external supervisors Carsten Maartmann-Moe and Antonio Martiradonna, at Ernst & Young AS, for their valuable contributions. We would also like to thank Danilo Gligorosky at the Department of Telematics for providing constructive feedback throughout the semester, and for giving us the opportunity to write this thesis. In addition, we would like to thank all contributors of code and documentation to the open source software Tahoe-LAFS, which has given us great inspiration and building blocks for our thesis.

Best regards,

Eirik Haver, Eivind Melvold and Pål Ruud

June, 2011

# Contents

# List of Figures

# List of Tables

# Listings

# Acronyms

| | |
|---|---|
| **ACL** | Access Control List |
| **AES** | Advanced Encryption Standard |
| **BFDA** | Brute Force and Dictionary Attack |
| **CA** | Certification Authority |
| **CBC** | Cipher Block Chaining |
| **CDA** | Cluster Dictionary Attack |
| **CSPRNG** | Cryptographically Secure Pseudorandom Number Generator |
| **CPU** | Central Processing Unit |
| **CSV** | Cloud Storage Vault |
| **CTR** | Counter |
| **DRY** | Don't Repeat Yourself |
| **DSA** | Digital Signature Algorithm |
| **DSS** | Digital Signature Scheme |
| **EC2** | Elastic Compute Cloud |
| **ECB** | Electronic Codebook |
| **GPU** | Graphics Processing Unit |
| **FEC** | Forward Error Correction |
| **HDFS** | Hadoop Distributed File System |
| **HTTP** | Hypertext Transfer Protocol |
| **HTTPS** | Hypertext Transfer Protocol Secure |
| **IaaS** | Infrastructure as a Service |

| | |
|---|---|
| **IV** | Initialization Vector |
| **JCA** | Java Cryptography Architecture |
| **JCE** | Java Cryptographic Extensions |
| **JVM** | Java Virtual Machine |
| **LAFS** | Least Authority File System |
| **MAC** | Message Authentication Code |
| **MITM** | Man-in-the-middle |
| **NIST** | National Institute of Standards and Technology |
| **P2P** | Peer-to-Peer |
| **PBKDF2** | Password-Based Key Derivation Function version 2 |
| **PaaS** | Platform as a Service |
| **PasS** | Privacy as a Service |
| **PEP** | Python Enhancement Proposal |
| **PGP** | Pretty Good Privacy |
| **PKI** | Public Key Infrastructure |
| **QR** | Quick Response |
| **RAM** | Random Access Memory |
| **ROM** | Read Only Memory |
| **REST** | Representational State Transfer |
| **RSA** | Rivest, Shamir and Adleman |
| **SaaS** | Software as a Service |
| **SDK** | Software Development Kit |
| **SHA** | Secure Hash Algorithm |
| **SQL** | Structured Query Language |
| **SSL** | Secure Socket Layer |
| **TCCP** | Trusted Cloud Computing Platform |
| **TCG** | Trusted Computing Group |
| **TLS** | Transport Layer Security |

| | |
|---|---|
| **TPM** | Trusted Platform Module |
| **TTP** | Trusted Third Party |
| **UMTS** | Universal Mobile Telecommunications System |
| **URI** | Uniform Resource Indetifier |
| **URL** | Uniform Resource Locator |
| **USB** | Universal Serial Bus |
| **VM** | Virtual Machine |
| **WPA** | Wi-Fi Protected Access |
| **WSGI** | Web Server Gateway Interface |

# 1

# Introduction

The term *Cloud Computing* is not yet clearly defined [1], but involves the provision of software or computational resources available by demand via the Internet. In a draft [2], the National Institute of Standards and Technology (NIST) defines cloud computing as:

> *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."*

More and more of the traditionally locally hosted services is moving to the cloud. The amount of flexibility and cost savings this provides, can be quite extensive.

Today, we take the services that the established cloud providers offer us for granted. For example, both Google and Microsoft provides the services of online document editing, email, file storage and more for free, and available at any time from anywhere [3, 4].

However, this often comes at a cost of reduced privacy, as the control over the hosting environment is lost. Users are therefore forced to increasingly think about how the data stored online can leak to unwanted people, either by accident, or by purpose by unfaithful servants. Should it be sufficient to trust the security policies of the providers, or is it possible to handle the privacy issues locally?

## 1.1 Motivation

In the last decade, the available supply for services that offer data storage remotely in the cloud has increased considerable. Storing private data at a third party provider, in contrast to self-hosted storage devices, has proven to be preferable due to low storage costs and high reliability, scalability and capacity.

However, storing data at a remote location prevents users from physically protecting their storage medium. With this in mind, there is no guarantee that the data of customers is kept private and secure from unfaithful servants at the storage provider.

To solve this issue, there has lately been introduced numerous applications and architectures [5, 6, 7] that ensure privacy and integrity of data stored at the provider. However, all of these alternative systems are missing one or more properties towards being, as we will define it, a completely secure cloud storage solution. This has motivated us to create a scheme that answers all of these challenges.

Additionally, we do also see this as a golden opportunity to learn more about software development, development methodologies, team work, and practical use of information security and cryptography.

## 1.2 Related Work

In the later years, there has been a lot of research done in the field of security in cloud computing. The problems that arise are fundamentally not different from those revealed by classic information security scenarios. The key point is that when using a service hosted by someone you do not know, you have to treat that *someone* as untrusted and as a possible attacker. Generally speaking, you loose control over the hosting environment, and hence has to deal with the security issues this implies.

In Section 2.5, we present four papers [7, 8, 6, 9] which try to solve security issues in a shared hosting environment. Common to all of these, is that they either rely on special, secure and tamper-proof hardware and/or a trusted third party.

Another way of providing a solution to the same problem, is to give the responsibility of the security operations to the client, i.e. in an environment that the user has control of. In Section 2.6, we present three publicly available software services that relates to this approach.

One of these applications, Tahoe-Least Authority File System (LAFS) [5], is given special attention. This is because it is an open source and a well documented piece of software, that answers most of the problems arising in an untrusted cloud storage environment, and relates closely to the work performed in this thesis.

## 1.3 Scope and Objectives

This thesis has two main objectives. The first objective is to create a cryptographic scheme that can provide secure storage of data by using an untrusted cloud storage provider. We define the criteria of a secure storage system by the following points:

1. It is possible to verify the integrity of the stored data.
2. Only authorized people can access the data.
3. Confidentiality is to be assured in a safe environment prior to storing data at the provider.
4. The storage scheme is documented in detail, such that users can easily understand the scheme and accept it on that basis.
5. An implementation of the storage scheme is open source.
6. An intrusion of the server does not affect the confidentiality of the data.
7. The use of a trusted third party is not mandatory.

The cryptographic scheme shall in addition provide the possibility of sharing stored data between multiple cloud storage users, while sustaining desired security. The underlying scheme for sharing data should further be applicable for both enterprise and personal user scenarios.

The second main objective is to implement the proposed scheme as part of a proof of concept application for Android devices. The application shall be designed for personal users and implement the most necessary features, and be measured in security and performance.

Finally, it is important that the implementation of the cryptographic scheme is available as open source.

## 1.4 Limitations

We will focus on making an architecture that covers the scope and objectives, in an easy to understand and thorough way. In addition, making core functionality, that demonstrates the most important security features in a proof of concept system, will be prioritized.

However, due to time and resource constraints, we will focus less on the following:

- The language in the proof of concept client should be clear, but the Graphical User Interface (GUI) itself will not be prioritized
- Experimentation with the proof of concept code, other than basic performance and security measurements
- Experimentation with the proof of concept client on hardware equipment other than what we easily have available at the time of testing

## 1.5 Methodology

The work behind this thesis, is carried out by the three authors in cooperation. The methodology used, can be categorized based on the three main parts of this work; the *research*, the *design and abstraction* part, and the *software development* cycles.

The research will include an analysis of related systems, and a study of relevant background theory. Based on this theory, we will use experimentation to create and design a theoretical solution to the problem of secure storage and sharing of files on an untrusted server. We will iteratively analyse our experimentation to find and correct flaws with the design.

The third part of the work, is the software development cycles of the proof of concept application.

**SCRUM** We will work after SCRUM principles – an iterative and incremental based framework for project management [10]. There does not exist a SCRUM *product owner* for the system we will create, nor do we fulfil the requirements and characteristics of a traditional SCRUM *team*[10]. Hence, we will use the principles that are practically possible for us to follow:

- Daily *stand-up* with planning of the tasks of the current day
- Weekly *sprint*[1] planning meetings
- Keep tasks on stickers, that we move between different phases on a board: *to do*, *in progress*, *quality assurance* and *done*. This is to keep track of progress in the current sprint
- Continuously analyse the process, and improve it if possible

**DRY** We choose to follow the Don't Repeat Yourself (DRY) principle when developing software. Hunt and Thomas [11] define DRY as:

> *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*

This principle can be taken further, as to not develop something that has already been developed in the past. If there exist a library for a given task that does fulfil the requirements set for the specific task, we will choose to utilize the library instead of developing similar code by ourselves.

## 1.6 Outline

This thesis is presented as per the following chapters:

[1] A *sprint* is a defined period with a given set of tasks.

**Chapter 2 – Background** provides background knowledge of the security services, technologies and software used to form a secure cloud storage system. In addition, relevant research and commercial solutions are scrutinized.

**Chapter 3 – Technical Procedure** goes through the development process of the scheme and software produced by this thesis. It starts with an overview of the architectural properties, followed by the more specific cryptographic scheme that fits in with the architecture. Lastly, the implementation of the proof of concept system is described.

**Chapter 4 – Experimental Procedure** presents the measurements and practical experimentation done to look at how the system behaves performance- and security-wise.

**Chapter 5 – Results** illustrates the findings from the experimentation.

**Chapter 6 – Discussion** reflects on the specific implementation and results from the previous chapters. Associated functionality to a secure cloud storage system is presented and discussed.

**Chapter 7 – Conclusion and future work** extracts the most important results and findings, and concludes the work done in this thesis. Further work that can be applied to the created system and scheme, are prioritized and presented as final ideas.

In addition, included appendices consist of:

**Appendix A – Other Relevant Implementations** presents the implementations created to carry out the security experiments.

**Appendix B – Attachments** describes the contents of the supplied attachments.

# 2

# Background

The basis and underlying technologies for a secure storage service in the cloud, are numerous and often complex. In the following sections, we will go through the security services, cryptographic primitives and attacks that are relevant. In addition, we present related research and existing solutions available at the time of writing.

## 2.1 Cloud Computing

In this section, we will extend from the definitions given in Chapter 1, and further describe terms that are associated with *Cloud Computing*.

### 2.1.1 Service Models

NIST defines three service models which deals with what kind of service the consumer can rent from a provider [2].

**Software as a Service (SaaS)** The capability of a consumer to run application of the provider on cloud infrastructure, using a thin-client, browser or similar, is called SaaS. The web-based email service GMail[1] can be seen as an example of this.

[1] `http://www.gmail.com/`

**Platform as a Service (PaaS)** The capability for a consumer to deploy software onto the cloud, but without actually controlling the underlying platform, operating system and so on, is called PaaS.

**Infrastructure as a Service (IaaS)** The capability provided to the consumer to provision processing, storage, networks and other fundamental computing resources where he can run arbitrary software, including operating systems and applications, is called IaaS. An example is when renting a Virtual Machine (VM).

### 2.1.2 Deployment Models

The NIST draft lists several deployment models which deals with how the cloud is organized in terms of where it is hosted, and who has access to it.

**Private Cloud** A private cloud is a cloud infrastructure operated solely for an organization. The party managing the cloud, and where it is located is not defined.

**Community Cloud** A community cloud is a cloud infrastructure shared by several organizations to serve a common concern. Where it is located, and who manages it, is not given.

**Public Cloud** A public cloud is a cloud infrastructure where everyone, or at least a large group, can have access, and is owned by an external provider of cloud services.

**Hybrid Cloud** A hybrid cloud is a cloud infrastructure composed of two or more clouds of any other model.

## 2.2 Security Services

This section explains the security services used in this thesis. A security service is any processing or communication service that enhances the security of the data processing systems and the information transfers of any organization, as defined by Stallings [12, p. 12].

**Confidentiality** Confidentiality is the act of keeping a message secret from unauthorized parties [12, p. 18]. This can typically be done by either preventing other parties access to the message at all, or by making the contents unreadable, for instance by the use of encryption.

**Integrity** Integrity implies that a message cannot be altered without the receiving part noticing. In a security perspective, integrity deals with detecting, preventing and recovering a message being changed by an attacker [12].

**Availability** The property of a system being accessible and usable upon demand by an authorized system entity, are defined by the availability service [12].

**Authentication** Authentication is the act of a user, service or similar to prove that he is what he claims to be [12].

**Non-Repudiation** Non-repudiation prevents both the sender and the receiver of a message from refuting the authenticity of transmitted message. In other words, one party can prove the involvement of the other party [12].

## 2.3 Security Attacks

This section briefly list security attacks relevant to this thesis, as defined by Stallings [12, Ch. 1.3].

**Active and Passive Attacks** Two general classifications of security attacks exist, where a *passive attack* attempts to learn or make use of information from the system, but does not affect system resources. An *active attack* attempts to alter system resources or affect their operation.

**Traffic Analysis** Traffic Analysis is the act of capturing and examining communication data sent between two parties. This information might contain secrets, or for instance leak enough information about an encryption key to recover it.

**Masquerade** Masquerade is an active attack where the attacker pretends to be one of the legitimate parties.

**Replay** Replay is an active attack where the attacker capture some data in a communication session and subsequently retransmit that information.

**Modification of Messages** Modification of messages is an active attack where the attacker alters some of the contents of a message sent between two communicating parties.

**Denial of Service** Denial of Service is an active attack where the attacker seeks to make resources unavailable for legit users, i.e. by overloading an application by sending a great amount of traffic.

**Man-in-the-middle** In the Man-in-the-middle (MITM) attack, an attacker intercepts messages between the communicating parties and then either relay or substitute the intercepted message. This an active attack.

### 2.3.1 Attacks on Cryptographic Primitives

Even though cryptographic primitives are designed to be secure, they might have implementation flaws and be used in an improper fashion, e.g. by using wrong parameters.

**Cryptanalysis Attack** A cryptanalysis attack is an attempt to deduce a specific plaintext or to deduce the key being used in a ciphertext.

**Brute-Force Attack** In a brute-force attack, an attacker tries to obtain a secret by testing the algorithm with up to all possible inputs. The secret might be an encryption key, or the data fed into a cryptographic hash function.

A related attack is the *dictionary attack*, where the attacker tries to obtain a secret by trying a subset of all known inputs, i.e. a predefined dictionary of words.

### 2.3.2 Security Considerations in Cloud Computing

There are some considerations when using cloud services from an external provider, as opposed to self controlled hardware, software and platforms. Most notably is that you loose the control of selecting the people which will have physical and digital access to the infrastructure [13]. In essence, this means that the provider can read every data sent to and from the cloud as well as the data saved in the cloud.

Another risk is that information might be leaked to other users of the same cloud. For instance it might be possible for a VM to leak information to other VMs on the same host [13]. The highest risk of this is in a public cloud where almost anyone can gain access.

## 2.4 Cryptographic Primitives and Applications

This section describes the low level security primitives and applications used throughout this thesis.

### 2.4.1 Randomness

Randomness is a basic property that multiple of the cryptographic primitives rely on, and hence deserves an explanation. Random data is informally defined as unpredictable to the attacker, even if he is taking active steps to defeat the randomness [14, p. 137].

A *Cryptographically Secure Pseudorandom Number Generator (CSPRNG)* produces pseudo random numbers based on a generated seed from a deterministic algorithm. What separates a CSPRNG from a PRNG is that it has properties making it suitable for use in cryptography. In other words, a CSPRNG should pass statistical randomness tests while simultaneously resisting serious attacks and predictions. Predictions should not be possible even if an attacker sees much of the random data generated by the CSPRNG [14, p. 140].

### 2.4.2 Encryption

Encryption is the process of transforming some information into an unreadable form for anyone not possessing a secret, *the key*. It is primarily used to enforce confidentiality, but can also be used for other purposes, e.g. authentication.

In its basic form, an encryption scheme consist of an encryption algorithm (the *cipher*), a key and a message (the *plaintext*), that is all used to create an encrypted message, i.e. the *ciphertext*. If a strong cipher is used, knowledge of the cipher, and multiple plaintext and ciphertext pairs, should not be enough to obtain the key, or to decrypt ciphertext with a corresponding unknown plaintext [14].

**Block Cipher and Stream Cipher** There are different classifications of how a cipher treats data [12, p. 32]. A *block cipher* will encrypt a block of data of a specific size. If the data is larger than the block size used by the application, a *mode of operation* is needed. In a *stream cipher*, the plaintext will usually be combined with a pseudorandom key stream to generate the ciphertext.

**Symmetric-key Encryption** A cipher where the same key is used for both encryption and decryption, is known as a symmetric-key algorithm [12, p. 32]. The Advanced Encryption Standard (AES) is a block cipher that works on a block of 128 bits, and support keys with length of 128, 192 and 256 bits. NIST standardized AES in 2001.

**The Mode of Operation** The mode of operation used for symmetric-key encryption enables subsequent safe use of the same key.

In a simple scenario, this could be to encrypt the normal data block-by-block with which is called the Electronic Codebook (ECB) mode of operation. The problem with this, is that some information of the plaintext will leak, i.e. the same plaintext will always be encrypted as the same ciphertext.

Another mode is *Cipher Block Chaining (CBC)*. In CBC, a non-predictable and unique Initialization Vector (IV) is used. The IV is XORed with the first block of plaintext, which again is encrypted with the cipher. The resulting ciphertext is used as an "IV" for the next block [12, p. 183], and so on.

**Asymmetric-key Encryption** Asymmetric-key encryption is an encryption scheme where different keys are used for encryption and decryption [12, p. 259].

An asymmetric-key encryption scheme is often called a *public-key encryption* scheme, where one key is defined as private and the other as public. The public key is shared to allow other parties to encrypt messages that only the owner of the private key can decrypt.

The downside of asymmetric compared to symmetric cipher is that it requires a larger key, and that it has a larger computational overhead to obtain the same level of confidentiality. The probably best known asymmetric cipher is Rivest, Shamir and Adleman (RSA).

### 2.4.3 Cryptographic Hash Functions

A cryptographic hash function is a deterministic mathematical procedure, which takes an arbitrary block of data and outputs a fixed size bit string. The output is referred to as the *hash value*, *message digest* or simply *digest*.

Another property of a cryptographic hash function, is that the smallest change in the input data, e.g. one bit, should completely change the output of the hash function. In other words, it should be infeasible to find the reverse of a cryptographic hash function [12, p. 335]. It should also be infeasible to find two blocks of data which produce the same hash value, i.e. a *collision*.

The standard for cryptographic hash functions today, are Secure Hash Algorithm (SHA)-1 and the SHA-2 family.

### 2.4.4 MAC Functions

Ferguson et al. [14] defines a Message Authentication Code (MAC) to be a construction that detects tampering with a message – i.e. it authenticates the message. A MAC can be constructed in different ways. One example is HMAC which constructs the MAC using a secret key, the message and a hash function [15].

### 2.4.5 Key Derivation Functions

A key derivation function is a function which takes a key, a password, a passphrase or similar, and creates a new key from it. One of the applications of such a function, is to create a stronger key from a weaker key, such as a password. This technique is called *key stretching*. The process involves making the derivation of a key from a password an expensive process in terms of computing power, which in turn makes it more resistant to brute force attacks.

Password-Based Key Derivation Function version 2 (PBKDF2) is a key derivation function that utilizes key stretching. It uses a password, together with a randomly generated salt and a pseudorandom function [16]. The function will combine these inputs in a specific way, and can repeat the process for a specified number of times, called the *iteration count*. A higher iteration count results in a stronger key. The salt provides defence against a precomputed collection of keys, i.e. a *rainbow table*, in the sense that it will make sure that a password will not derive the same key if different salts are used.

### 2.4.6 Digital Signatures

A digital signature is the digital equivalent of a normal signature, i.e. it verifies that an entity approves with or has written a message. It can also verify the date the signature was made. In addition, it should be verifiable by a third party [12, p. 379]. A digital signature should not be feasible to fake.

The RSA cipher can be used to generate signatures. In addition, there is also a standard for digital signatures, called Digital Signature Scheme (DSS), which uses Digital Signature Algorithm (DSA) as the underlying algorithm.

### 2.4.7 Digital Certificates and PKI

A digital certificate is the pairing of a digital signature and a public key [12]. By this scheme, the services confidentiality, authentication and non-repudiation can be achieved.

For example, a person has a certificate with some clues about an identity in it, e.g. an e-mail, together with a public key. This certificate can then be signed using digital signatures, to verify that some other entity trusts this certificate. In practice, the entity which signs certificates is the Certification Authority (CA), which all clients have the public key information for, and trusts. The CA will also contain information about which certificates has been revoked, i.e. should not be trusted in use. Such a scheme is usually referred to as a Public Key Infrastructure (PKI).

#### PGP

Pretty Good Privacy (PGP) is a scheme similar to PKI, but with no CA that all users trust [12]. Instead, trust is made between users by somehow verifying their public key, for instance by meeting face to face. A user can then sign the key of another user, set a trust level for the user, and publish this information to a key server. Other users can then calculate a trust on an unknown person, based on the trust set by people that they trust, from information located on publicly available key servers.

### 2.4.8 SSL/TLS

Transport Layer Security (TLS), and its predecessor Secure Socket Layer (SSL), are technologies for obtaining confidentiality, integrity and authentication for transfer of files over a network [12]. It does so by a combination of different algorithms and primitives, but a digital certificate is required for authentication.

To transfer files securely over Hypertext Transfer Protocol (HTTP), TLS/SSL is used to form Hypertext Transfer Protocol Secure (HTTPS).

## 2.5 Research on Security in Cloud Computing

This section will elaborate on selected research concerning privacy within cloud computing. The scrutiny on this subject can be divided into proactive solutions that either reduce, or prevent the risk of leaking privacy related information. Only the latter type of solutions are further discussed.

We choose to present these papers as they provide possible solutions to the same problems handled in this thesis, although with a different approach.

A criterion for secure cloud storage, as described in Section 1.3, is that unauthorized access to the encrypted files stored on the server should be avoided, but that it should not be crucial for the security of the content of the files.

The following first three security systems seek to secure the cloud server itself, and the last one argues for building a secure system on top of non-trusted cloud servers.

### 2.5.1 Privacy as a Service

A concept entitled Privacy as a Service (PasS), was suggested in 2009 [7]. PasS is a set of security protocols ensuring privacy of customer data in cloud computing architectures. The main design goal with PasS, is to maximize the user's control over his sensitive data, both processed and stored within a cloud.

The PasS concept is based on a fundamental *system model* and *trust model*. The system model consists of three communicating parties, namely a *cloud provider*, a *cloud customer* and a *Trusted Third Party (TTP)*. The PasS system model is shown in Figure 2.1.

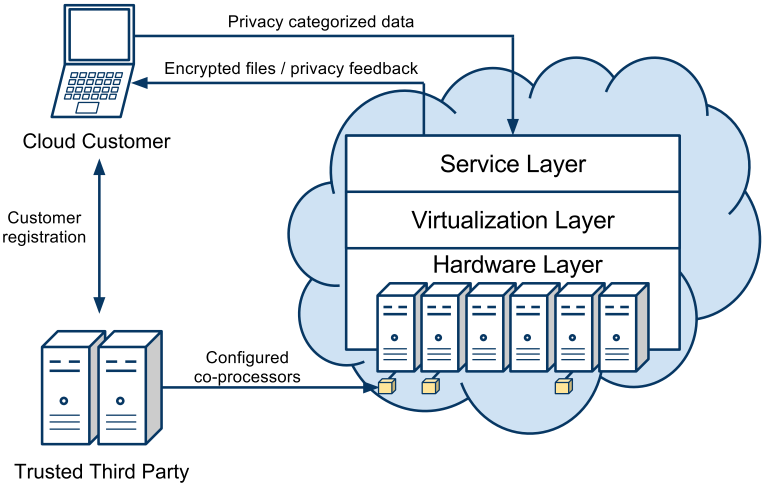


Figure 2.1: System model of PasS

It is important to notice that the PasS system model is dependent on pre-installed cryptographic coprocessors in the hardware running the cloud service. A cryptographic coprocessor is a small hardware card, including a processor, Random Access Memory (RAM), Read Only Memory (ROM), backup battery, persistent storage and an Ethernet network card. A coprocessor interfaces with a server in the cloud, and provides a safe environment for processing of customer's data

The cryptographic coprocessors are used in the cloud because they are supposedly tamper-proof against physical attacks. The coprocessors are pre-configured by the TTP before they are installed. By using this procedure,a safe computational environment for the cloud customer is provided, which is kept secret from the cloud provider.

The main task of the TTP is to compute a set of public/private key pairs, load them into the persistent storage of the co-processor, and further send them to the customer. The TTP also loads its own secret key into the coprocessor. This key distribution ensures secure communication between the three parties. The key pair of the customer is sent through a secure communication channel.

With cryptographic coprocessors in the cloud and a secure communication, the cloud customer can choose between three different levels of privacy towards the cloud provider – no privacy, privacy with a trusted provider and privacy with a non-trusted provider.

*No privacy* implies storing data as clear text in the cloud. *Privacy with a trusted provider* involves storing encrypted data in the cloud. This data is encrypted by the cloud provider and only achievable by the customer or cloud provider.

In the case of *privacy with a non-trusted provider*, the customer encrypts the private data before uploading it to the cloud provider. The key used for encryption is shared with the cryptographic co-processor, through an authenticated version of the Diffie-Hellman key management protocol. The co-processor can further process the encrypted data and store it in the cloud facility. The stored data is encrypted and unknown to the cloud provider.

### 2.5.2 Privacy Manager

In 2009, HP Labs proposed a way to manage and control the private data of users, stored and processed in a cloud facility [8]. Their solution was partially implemented as a software program called a *privacy manager*.

The privacy manager uses a feature called *obfuscation*, which is quite similar to encryption. However, the obfuscation method is different from encryption in the sense that the obfuscated data can be processed in the cloud, without the cloud provider knowing the encryption key or the original data. Pearson et al. [8] mention the following obfuscation methods:

- Yao's protocol for secure two-party computation [17]
- Gentry's homomorphic encryption scheme [18]
- Narayanan and Schmatikov's obfuscation method [19]

Due to better efficiency, the privacy manager uses the latter alternative. However, Narayanan and Schmatikov's obfuscation method does not provide complete confidentiality to the cloud provider [19].

In addition to installing a privacy manager at the user's terminal, HP Labs suggests the use of trusted computing solutions to address the lower-level protection of data. The Trusted Computing Group (TCG)[2] is an example of an organization developing and providing trusted computing solutions. A tamper-proof piece of hardware called a Trusted Platform Module (TPM) is recommended [8], which is designed by TCG. The TPM is installed in the machine running the privacy manager, to ensure that processes carried out by the privacy manager can be fully trusted.

The privacy manager is suggested to work in three different use cases. It can be implemented to support a *single client*, the use of *hybrid clouds* and/or the use of an *infomediary* within the cloud.

[2] `http://www.trustedcomputinggroup.org/`

### 2.5.3 Trusted Cloud Computing Platform

Equal to Privacy as a Service and the privacy manager, *Trusted Cloud Computing Platform (TCCP)* was proposed as a solution to provide secure computations and storage within a non-trusted cloud provider [6]. As opposed to the previous solutions, TCCP is directed against secure execution of guest VMs outsourced to IaaS providers.

The original infrastructure, before adding TCCP, is assumed to consist of a *cloud manager*, which manages a cluster of nodes running one or more VMs. Among multiple tasks, the cloud manager is responsible for loading VM images into its own nodes. Each node has a *VM monitor* which will further launch and monitor VMs from the received corresponding images.

TCCP is based upon the TPM chip and is a *remote attestation scheme*. The scheme enables a network entity to verify whether another remote entity runs a TPM chip or not.

The TCCP system architecture is illustrated in Figure 2.2. The trusted computing base of TCCP includes a *trusted coordinator* and a *trusted virtual machine monitor*. The coordinator manages the trusted nodes within a cluster. To be trusted, a node must be located within a security perimeter and run a trusted virtual machine monitor.

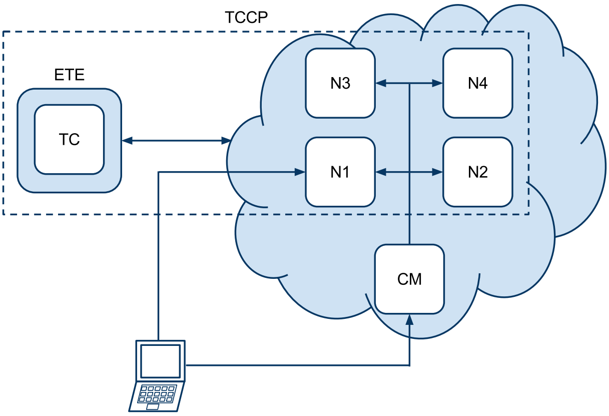


Figure 2.2: System architecture of TCCP

The coordinator maintains a record of the nodes located in the security perimeter, and use remote attestation to ensure nodes are trusted. Each trusted node in a cluster contains a TPM chip and a corresponding trusted monitor. The main task of the trusted monitor, is to enforce a local closed box protection of a client's running VM.

Each trusted virtual machine monitor cooperates with a trusted coordinator to protect the transmission of VMs between trusted nodes, and to ensure that VMs are executed by trusted nodes. In this context, the TCCP specifies several protocols for both launching and migrating VMs inside the cloud. These protocols are described by Santos et al. [6].

The trusted coordinator-part is installed in servers operated and maintained by a trusted third party, to prevent unwanted tampering from the IaaS provider. A client can further use remote attestation to the coordinator to verify that the IaaS provider secures its computation.

With TCCP, the client interacts with the IaaS provider as usual. The difference is that the trusted nodes and their trusted coordinator communicates to ensure a secure environment for executing the client's VM.

### 2.5.4 Cryptographic Cloud Storage

In 2010, researchers at Microsoft were looking at the problem of building a secure cloud storage service on top of a non-trusted cloud storage provider [9]. They described architectural solutions related to both consumers and enterprises. The architectures were explained in high level and were designed to utilize and combine recent and non-standard cryptographic primitives.

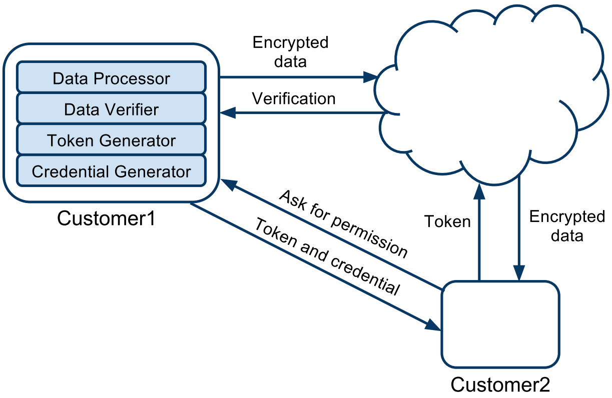


Figure 2.3: Cryptographic cloud storage, customer scenario.

The consumer architecture is depicted in Figure 2.3, and a typical enterprise architecture is shown in Figure 2.4.

Each architecture consists of the following computational components:

**Data Processor**
Processes data before it is sent to the cloud.

**Data Verifier**
Checks whether data stored in the cloud has been tampered with.

**Token Generator**
Generates tokens that enable the cloud provider to retrieve segments of customer data.

**Credential Generator**
Responsible for creating and distributing access policies.

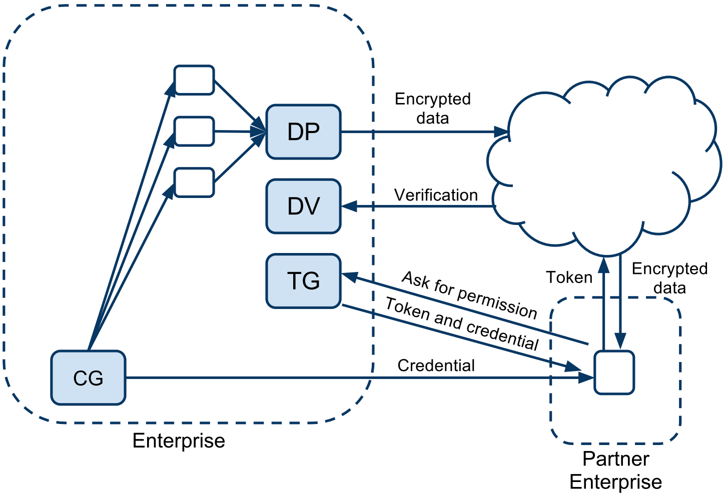


Figure 2.4: Cryptographic cloud storage, enterprise scenario.

The core components are suggested to support *searchable encryption*, *attribute-based encryption* and a *proofs of storage* protocol.

## 2.6 Existing Solutions

There are a number of existing solutions for storing data in the cloud, with more or less the functionality required to fulfil the problem description for this thesis. The section highlights some of them.

### 2.6.1 Dropbox

Dropbox[3] is a popular commercial application for storing data in the cloud, claiming more than 25 million users [20]. All files are saved using Amazons S3 storage service.

The company boasts of strong encryption and strict access control [21], but has received criticism for its lack of security [22]. Among these concerns, is the *Forgotten Password* feature, that enables Dropbox to hold the passwords of their users. This implies that the encryption is performed server-side and that Dropbox can read all data stored with their service.

In addition, Dropbox is not open source, and hence it is difficult to verify that the security features actually work as claimed.

[3] http://www.dropbox.com/

### 2.6.2 Tahoe-LAFS

Tahoe-LAFS[4] is an open source, distributed and secure cloud storage file system, fulfilling the criteria in Section 1.3. The integrity and confidentiality of the files are guaranteed by the algorithms used on the client, and is independent of the storage servers, which may be operated by untrusted people. This is defined as *provider-independent security* [5].

In Tahoe-LAFS, files are exclusively encrypted client-side, then split up using *erasure coding*, before being uploaded to the cloud, as illustrated in Figure 2.5.

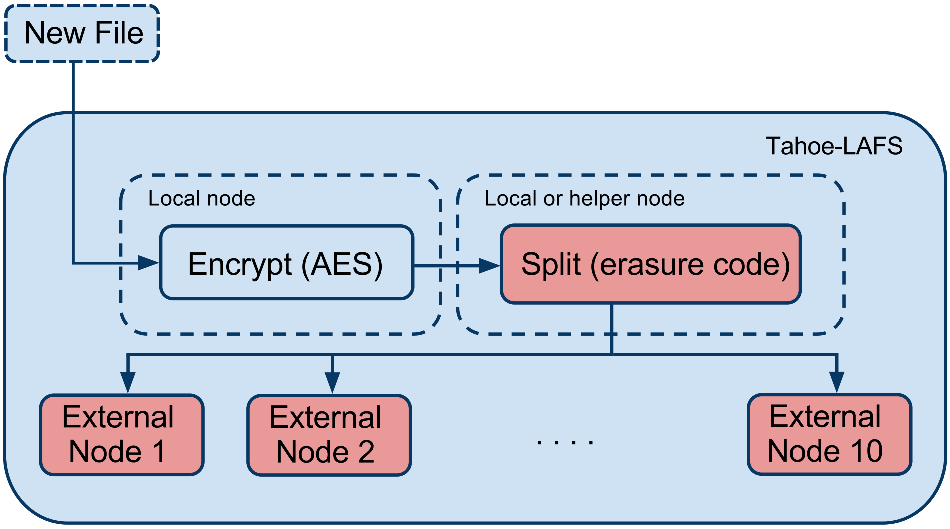


Figure 2.5: Tahoe-LAFS: Insertion of a new file

#### Architecture of Tahoe-LAFS

Tahoe-LAFS has a three layer architecture: the key-value store, the file system, and the application [5].

The **key-value store**, is the lowest layer and is implemented by a grid of Tahoe-LAFS storage servers. Data is kept on the storage servers in the form of *shares*, which are encrypted and encoded parts of files. Capabilities are short ASCII strings, containing information on where to find, decrypt and verify a file or folder. Nodes in the grid learn about each other through an *introducer*.

The **file system** layer is responsible for mapping human-meaningful pathnames to pieces of data. Each directory contains a table of capabilities for its children, i.e. subdirectories or files. Two main forms of capabilities are available for each file, read-only and read-write, and these can be distributed to e.g. share a file with friends.

[4] `http://www.tahoe-lafs.org/`

Since it is not practical for users to remember strings containing random characters, the **application** layer is used for providing a user-friendly interface to the directories and files.

**File Types** There are two kinds of files in the Tahoe-LAFS – **immutable** and **mutable** files. An immutable file is created exactly once. Mutable files can be modified, and everyone who has access to the signing key can make new versions of the mutable file. Directories are implemented as mutable files.

**Erasure Coding** Erasure coding with the Solomon-Reed scheme, enables Tahoe-LAFS to recover a file using only a predefined subset of the parts distributed to the storage servers. Erasure coding is a type of Forward Error Correction (FEC) code, which extends a message with $C$ characters into a longer message with $N$ symbols [23]. The original $C$ characters can then be recovered from a subset of the $N$ symbols.

**Sharing** To share a folder, and hence its subfolders and files, the corresponding capability of the folder has to be distributed. Tahoe-LAFS in itself does not provide a specific way of doing this, and leaves it up to the user to distribute keys in a secure manner.

### 2.6.3 Wuala

Wuala[5] is a software offering a secure cloud storage file system. It is written in the Java programming language, and hence has easily been ported to a number of platforms, e.g. Windows, Linux, OS X and Android.

**Sources of Information** The authors have released a paper on a cryptographic tree structure for the file system that Wuala uses, called Cryptree [24], but other details of how the system works is hard to come by. The only source of technical information of this system found, was a Google Tech Talk [25].

A side from this, Wuala is closed source, and hence no one can easily verify that the software indeed does what it states.

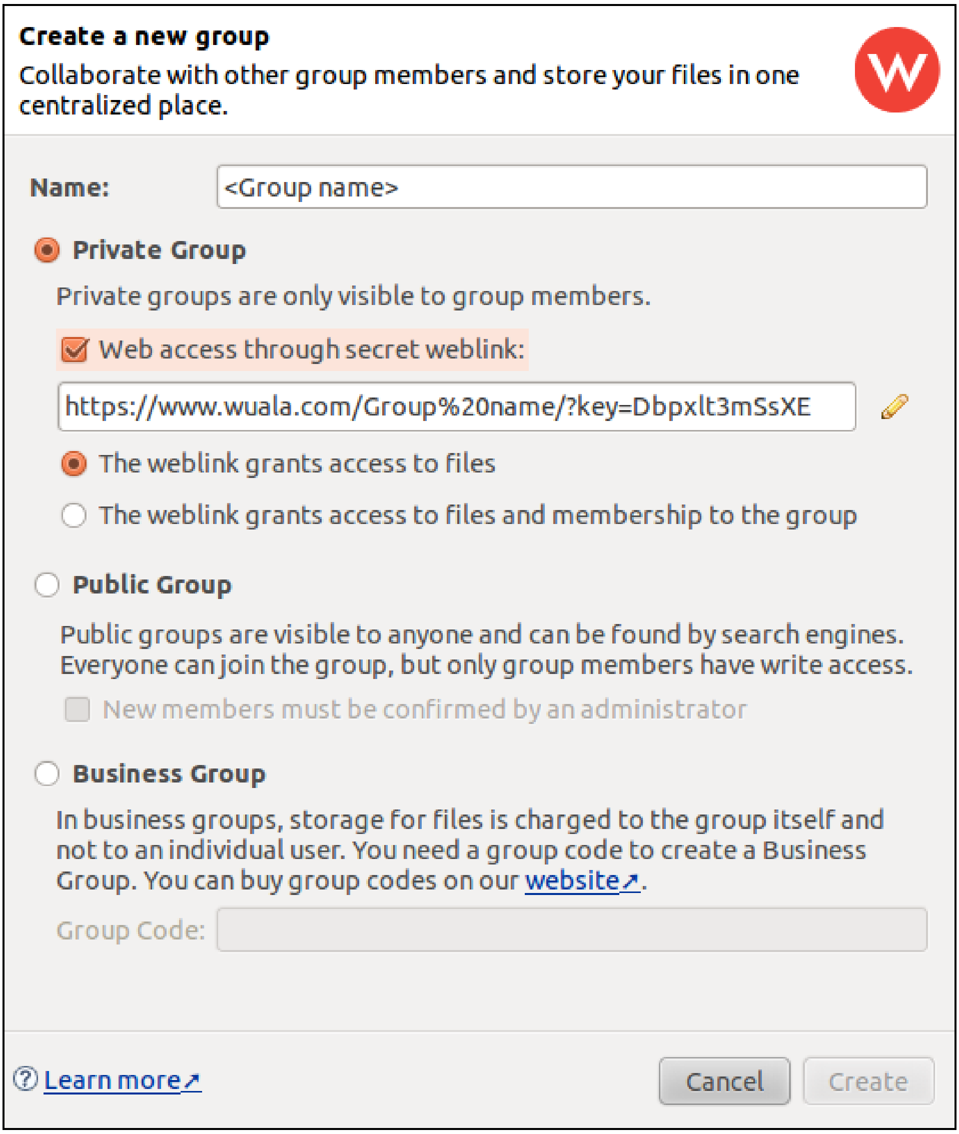
**Network Scheme** Wuala claims strong focus on reliability and availability, by both providing storage on their own central servers, in addition to a *Peer-to-Peer (P2P) cloud* of Wuala users that has donated capacity to the system. There are also additional advantages resulting directly from using a distribution scheme based on P2P. Examples of this are no maximum file size and no traffic limit.

Similar to the Tahoe-LAFS, Wuala uses an erasure coding scheme in the family of Reed-Solomon [25] to enable the logic behind splitting and combining parts of a file and creating redundancy.

[5] `http://www.wuala.com/`

**Sharing** Files and folders can be either *private*, *shared* or *public*. In addition, there exist a concept of public and private *groups* of users, which can be used to manage access control over shared folders and files.

When creating a new group in the Wuala client, as depicted in Figure 2.6, the default choice is to create a private group, but provide access through a *secret link* via the Wuala web page. This implies a key distribution where the group members has to rely on Wuala as a trusted third party.



**Create a new group**

Collaborate with other group members and store your files in one centralized place.

Name: <Group name>

● **Private Group**

Private groups are only visible to group members.

☑ Web access through secret weblink:

https://www.wuala.com/Group%20name/?key=Dbpxlt3mSsXE

● The weblink grants access to files

○ The weblink grants access to files and membership to the group

○ **Public Group**

Public groups are visible to anyone and can be found by search engines. Everyone can join the group, but only group members have write access.

□ New members must be confirmed by an administrator

○ **Business Group**

In business groups, storage for files is charged to the group itself and not to an individual user. You need a group code to create a Business Group. You can buy group codes on our website↗.

Group Code:

Figure 2.6: Creating a new group in Wuala

**Security** When adding a new file to Wuala, the file and its meta data are encrypted with 128 bit AES, before encoded into redundant fragments using erasure codes, and lastly uploaded to the network. 2048 bit RSA is used for authentication. All cryptographic operations are performed locally on the client-side, and the password used never leaves the client.

Access control are provided using the efficient cryptographic tree structure Cryptree [24, 25], which is based on the notion that no information should be revealed to the computer holding the access control structure, i.e. the hosting server or the P2P network. Keys for nodes in the tree are derived from the keys of the parents, implying that if a user has a key to a folder, he also has access to all the subfolders and files.

However, since the source of Wuala is not available for scrutiny by the public, none of these security features can easily be verified.

# 3

# Technical Procedure

This chapter will describe our original architectural and cryptographic scheme for providing secure storage of data on a remote untrusted system. It will further explain the procedures carried out to create a proof of concept application, that implements the proposed architectural and cryptographic scheme. The application, named *Cloud Storage Vault (CSV)*, is implemented as an Android application, and consists of separate server and client functionality.

The chapter will start by giving an overview of the proposed architecture followed by a more detailed description of the corresponding cryptographic scheme. The chapter will end by describing the implementation details for both the server and client side functionality of the Cloud Storage Vault.

## 3.1 Architectural Overview

The architectural solution of a secure cloud file sharing system has to convince its users that the functions indeed are secure, and that the concepts are easy to understand and accept. The following sections will elaborate on the architecture, favouring simplicity and familiar concepts, such as files and directories. We also introduce the concept of *capabilities*.

Figure 3.1 represents an overview of the functionality that the architecture must support. The illustration exhibits a user uploading a file to the cloud, and adding this to a parent directory. After he has done this, it is possible for him to distribute the capability of the file to other users to realize sharing of files or directories.

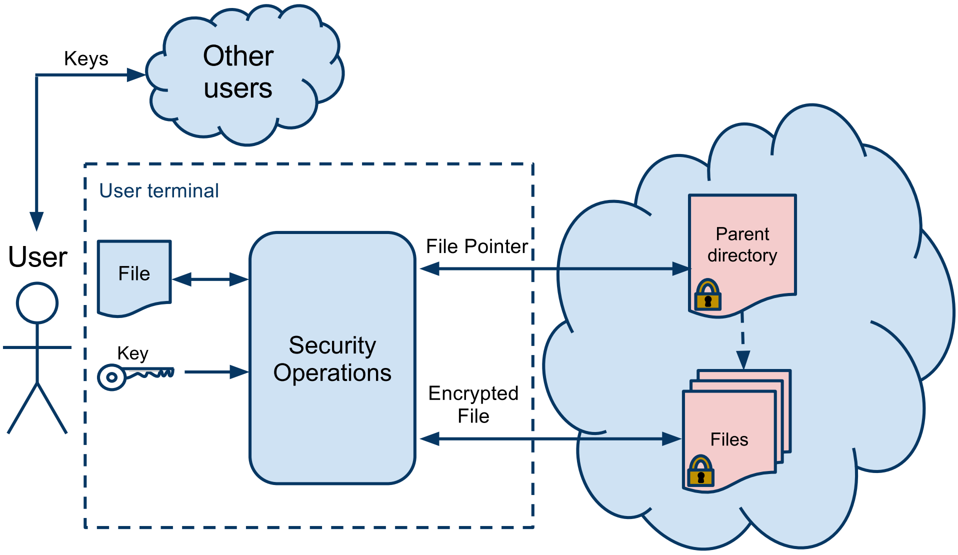


Figure 3.1: Overview of user functionality

### 3.1.1 File Storage

The solution for file storage proposed in this thesis, is that only a simple *key-value store* is needed on the server side. The key works as a lookup index for a specific value, while the corresponding value equals an encrypted file object. The server will be required to support the operations of uploading and downloading key-value pairs to this store.

From the users perspective, a file object can have multiple forms – it can either be a *mutable* or an *immutable* file. A mutable file can be changed, and is what a user will see as a directory, while an immutable file is as a normal file but cannot be changed.

A user will need certain information to be able to reach and read a file object, and we define these properties as the *capability* of a file object. For now, the capability represents the ability to find, read, verify that a file has not been tampered with, and write to a mutable file.

#### Directory Structure

The contents of a directory are files and other directories. More specifically, a directory contains the means to find files or directories, namely the corresponding capabilities. In addition, there exist a human readable name, an *alias*, for each entry in a directory. This design gives a flexible and space-conservative structure, since any file object may be found in multiple directories, but does only exist once in the cloud.

A user will have to have some way of storing the capabilities of his file objects. This could potentially be done client side, but a problem arises if the user wants to use several terminals. Thus, we introduce the *root folder*, a folder from which all other files and folders can be reached. The user will only need to know of one single capability to reach all his stored data. This capability has to be stored in a secure manner, e.g. in an *encrypted keyring*. The resulting structure is a directed graph, as illustrated in Figure 3.2.

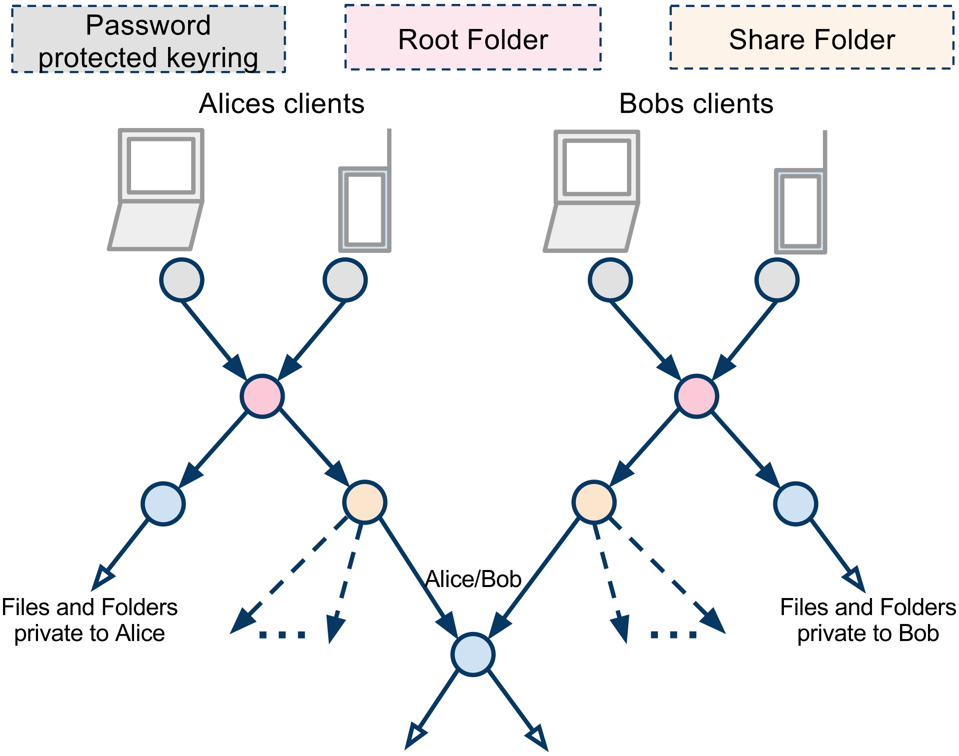

Password protected keyring
Root Folder
Share Folder
Alices clients
Bobs clients
Alice/Bob
Files and Folders private to Alice
Files and Folders private to Bob

Figure 3.2: File system structure

### 3.1.2 Authorization, Authentication and Accounting Layer

The possession of a capability gives a user access to read a file or read or write a folder, and hence serves as the primary access control. There are however some properties that the server provider might want that can not be given by the capability. Therefore a layer implementing authentication, accounting and authorization might be preferable.

#### Block Access to Encrypted Data

The capability for a file or folder might be intentionally or unintentionally leaked by a user. In this case, it would be preferable that the server can block access to a particular object. The server could also potentially enforce access rights on all encrypted objects, so that they are only retrievable by the owner. This would however complicate sharing.

#### Modification and Deletion of Files

For each directory, there exist a different capability for read and write operations, though the read capability can be deduced from the write capability. From the write capability, it is possible to deduce another secret, the *write enabler*, which the server also knows of. Knowledge of the write enabler is needed for the server to grant access to modify or possibly delete a folder.

For immutable files, there is no concept of write access, only read. A user might not want to pay for storage of files that he no longer needs. A layer that identifies the creators of a file, can by the same method decide who should have the rights to delete it.

#### Accounting

If the server side of the system is held by a cloud storage provider, it is important to be able to decide which users should be billed for the *file storage* and generated *network traffic*.

In the case of an immutable file, the storage costs can be billed to the user creating the file. The costs of network traffic can further be charged to the users retrieving the file.

Accounting might also be interesting for an organization using a third party cloud provider. For instance an employee who leaves the organization, might be tempted to copy all the data stored on the server. The organization should then be able to discover what has been done, using some form of an audit trail.

It is however worth noting that if the accounting happens server side, there is no real way to verify that all logs stored there are correct, since the cloud provider will have access to modify or delete them.

### 3.1.3 User Scenarios

The various user scenarios supported by the software, provides a logical way to describe the external properties of the system. The fundamental operations are *downloading*, *uploading* and *sharing* of files.

#### Download File

The download procedure is depicted in Figure 3.3. The client sends a download request with the identifier of a folder, which he possesses the capability of. The server will respond with the encrypted directory. The user will use the capability to decrypt the directory.

In the directory, the user finds the aliases and necessary capabilities to gain access to the children of that folder. If the user now wants to download a file from the accessed folder, he obtains the identifier from the capability, and requests the

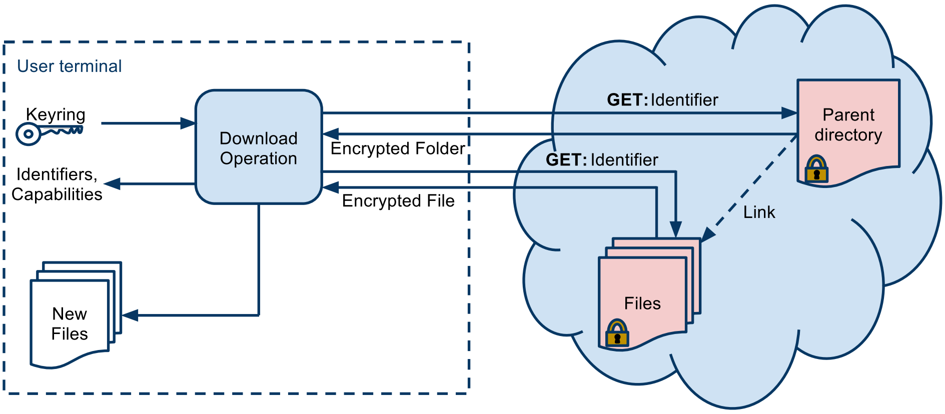


Figure 3.3: Scenario: Downloading a file

server for this file. Once downloaded, the capability provides means of decrypting and verifying that the data has not been tampered with.

### Upload File

Figure 3.4 shows the process of uploading a new file. The capability is generated by the client, and used to encrypt the data. The file is then uploaded to the server, and the capability and an alias is linked in to the parent folder.

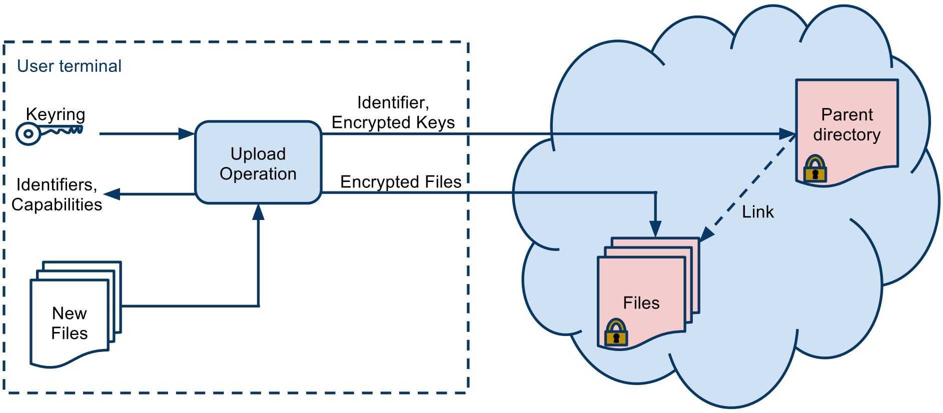


Figure 3.4: Scenario: Uploading a file

### Share Files

As shown in Figure 3.5, for Alice to be able to share files with Bob, she first has to create a new directory that will contain these files. Alice is then required to share the capability of the new directory with Bob. When the capability is shared, the new directory will work as a secure channel where Bob and Alice can share their own folders and files.

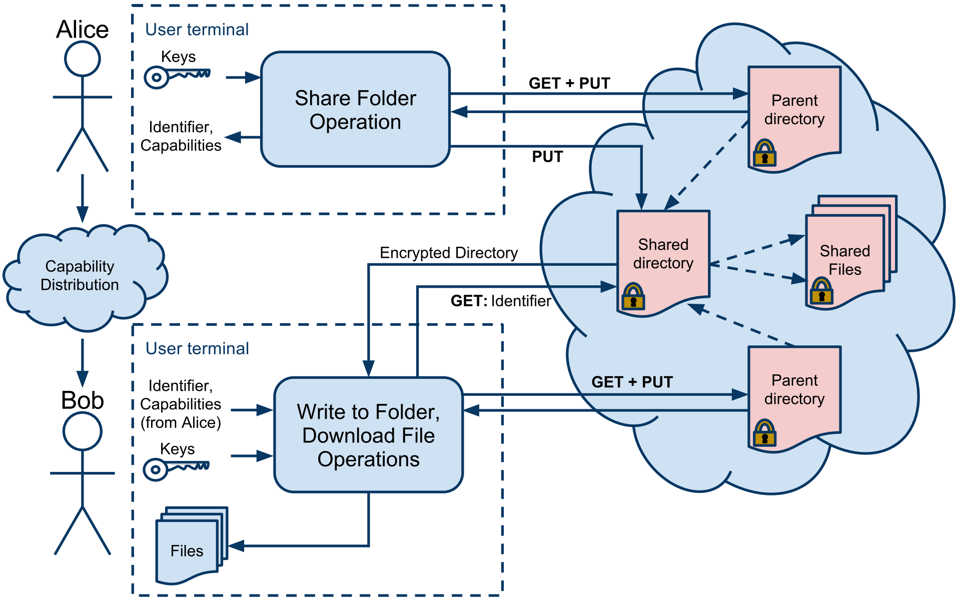


Figure 3.5: Scenario: Alice shares a file with Bob

Before transferring the capability to Bob, Alice links the shared directory to a parent directory, so she can easily retrieve it at a later time. She can also link files and other directories to the shared directory.

The capability distribution is a key design issue, and has to be performed in a secure manner. This can be solved in a variety of ways, and the solutions proposed in this thesis are discussed in Section 6.1.2.

After receiving the capabilities for the shared folder from Alice, Bob requests and receives the encrypted shared directory, in addition to linking it with a parent directory for future usage. He can then download shared files as if they where his own.

**Read-Only Shares** If Alice wants to share a directory in Read-Only mode, she can simply share the read capability with Bob, instead of the write capability. This will work as intended, but might prove somewhat cumbersome for Alice. If Alice wants to write to the directory she has shared with Bob, she cannot enter it through the parent folder shared with Bob, since this will only grant her the read capability. The implication is that Alice will have to access the directory through another path in her directory tree, to get the write capability.

A more simple solution is to enable Alice to store the write capability individually among her private files, while storing the read capability in the shared parent directory. The solution can easily be implemented by using a specialized *write key folder* under Alice's root folder. The write key folder will then contain write capabilities to every folder that Alice has shared in Read-Only mode. The idea behind the write key folder is illustrated in Figure 3.6.

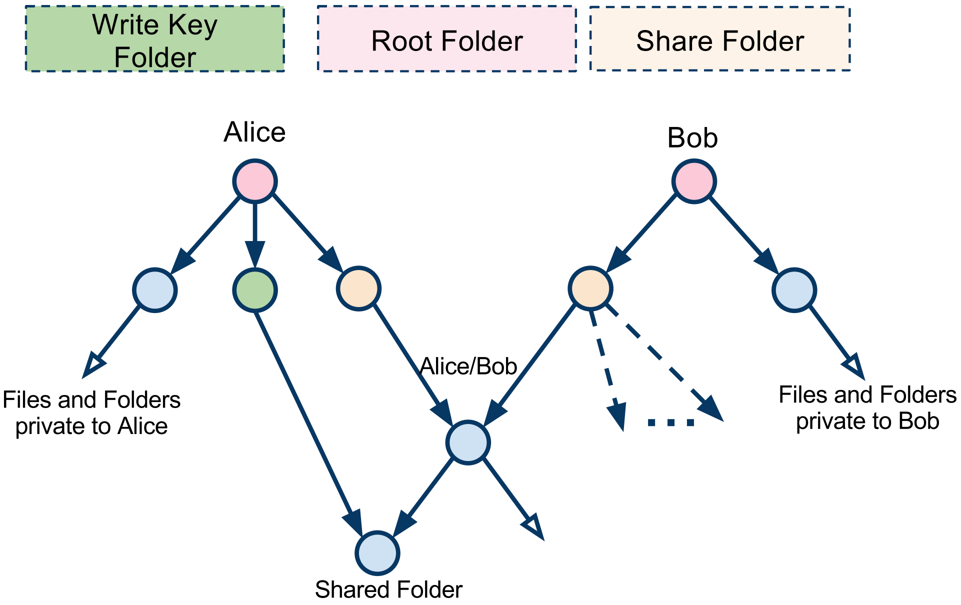


Figure 3.6: Sharing read only folders

### 3.1.4 Constraints

The software using this architecture should be able to run on restricted devices, i.e. equipment with limited memory and Central Processing Unit (CPU) power, often in addition to constraints on power and network utilization. This has implications for the design of the software, since all cryptographic operations has to be performed client-side.

## 3.2 Cryptographic Architecture

This section elaborates on the cryptographic solutions applied to the architecture in Section 3.1. It will take a closer look at how confidentiality and integrity are solved.

We will start with a brief introduction explaining the fundamental security concepts. The cryptographic architecture is further described in terms of file and directory operations. Key concepts are based on equivalent operations found in Tahoe-LAFS [5].

Table 3.1: The contents of a Capability

| **Data** | **Comment** |
|---|---|
| Type | A Capability is either read-only or read-write |
| Key | A cryptographic key |
| Verify | A hash needed to verify the contents of a file or folder |

### 3.2.1 Security Concepts

The basic security concept of the application is to keep the files of a user confidential to a third-party storage provider. To solve this, the application encrypts data locally at the user terminal before uploading them.

When accessing a file, the application downloads the encrypted file before decrypting it locally. To enable this simple encryption scheme, the user is required to possess the knowledge of at least one capability, which contains cryptographic keys to decrypt and verify the contents of the *root folder*.

The root folder will in turn contain the capabilities for its own children, which enables the client to decrypt files and folders stored in the root folder. The other folders work in the same manner.

By initially knowing that files are placed encrypted on a remote server and that the user possesses one or more cryptographic keys locally, we can continue with a more comprehensive description of the complete cryptographic solution. The details are explained in terms of capabilities and the operations conducted on files and folders.

**Capabilities** Capabilities are containers which contain the necessary information to locate, encrypt, decrypt, verify and possibly write file objects. The possession of a capability grants these rights, which can be either read access or read and write access. Such an access scheme is known as *capabilities as keys* [26]. The contents of a capability is summarized in Table 3.1, and will vary somewhat for files and folders, since they are implemented by immutable and mutable files respectively.

**Encrypted Keychain** Every user will need to keep a copy of his root capability somewhere. The capability contains both an encryption key and verification data that a user will need to safely access his files. This amount of random data will be hard for a person to remember, and therefore the capability will have to be stored on the terminal which runs the client software. Since it might happen that this terminal is either broken in to, lost or stolen, precautions should be taken to protect the capability on the device. The normal approach to this, is by using password protection.

The strength of a password is related to its length and randomness properties. Passwords shorter than 10 characters are usually considered to be weak [27]. In the event of loosing one of the clients, and thus the encrypted keychain, a potential attacker can use fast password cracking attacks to try to compromise the root folder keys. As an precaution for this, we will use PBKDF2/RFC2898[1] with a salt value, to create additional computational work for the process of unlocking the keychain. This method is known as key stretching [28].

### 3.2.2 File and Directory Operations

This section describes the elementary file and directory operations supported by the application. The basic file operations are *upload file* and *download file*, and correspondingly for directories, *create directory*, *open directory* and *modify directory.*

For simplicity, the illustrations in the following sections includes naming of cryptographic primitives. However, it is important to note that this cryptographic scheme will work with other primitives. Any symmetric cipher could work instead of AES, any signing function that uses both a private and public key could be used instead of RSA, any MAC could be used instead of HMAC-SHA and any cryptographic hash function could be used.

Though, the security of the system does rely on these choices, and a recommendation with rationale for each of the needed primitives will be given in Section 3.2.3.

#### Upload File

The operation behind uploading a file, is depicted in Figure 3.7. A random symmetric encryption key is generated. This is hashed once to obtain the *storage index* for the file. The storage index is then hashed again to form the IV for the file.

Next, the file is hashed and the resulting digest is stored together with the encryption key in the capability. Lastly, the file is encrypted with the encryption key and the IV, before transferred to the server.

#### Download File

The process of retrieving a file is illustrated in Figure 3.8. The storage index is obtained by hashing the stored encryption key extracted from the capability, and the IV is obtained by hashing the storage index. The file is then downloaded from the server and decrypted. Next, the file is hashed, and the resulting digest is compared against the digest stored in the capability. If these two match, the file has not been tampered with.

#### Create Directory

Creating and uploading a directory are illustrated in Figure 3.9. The process is a bit more complex than for files, because it has to support changing the contents of the folder.

Firstly, an asymmetric key pair is generated, and forms the private *signing key* and the public key. The signing key is hashed to form the *write key*, and again to form the *read key*, and once more to form the *storage index.*

[1] http://tools.ietf.org/html/rfc2898

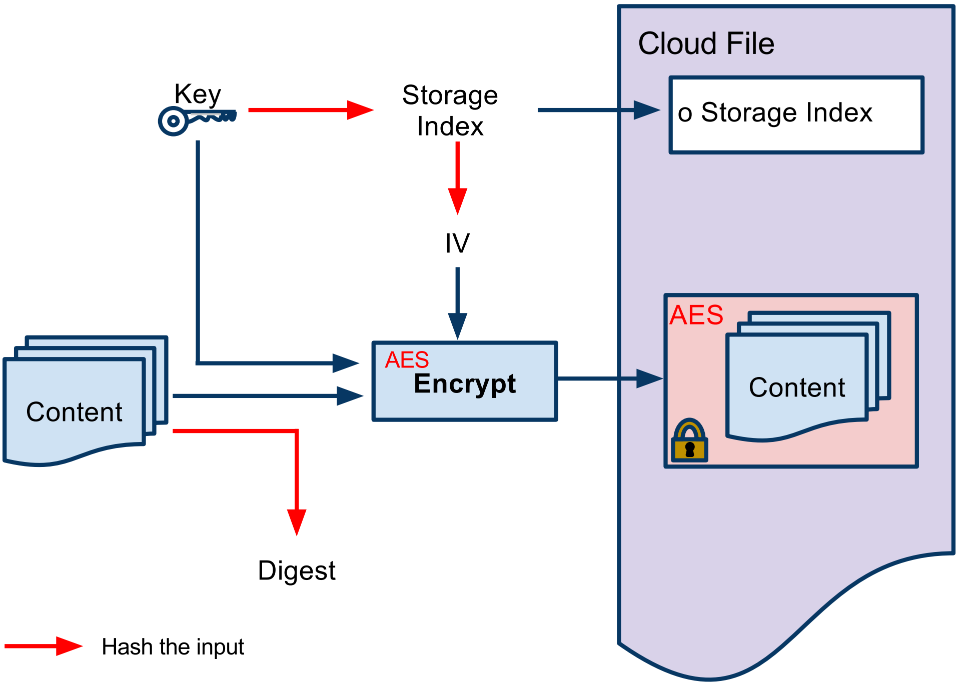


Figure 3.7: Behind the scenes: Uploading a file

The contents of the folder, which may be empty, is encrypted with the read key and the resulting ciphertext is signed with the signing key. The signing key is further encrypted with the read key, and together with the ciphertext, the public key, the IV and the signature, uploaded to the server.

The *write enabler* is deduced from the write key with the use of a MAC function and a fixed message. It is transferred alongside the directory. The write key is stored together with a hash of the public key in the capability.

### Open Directory

Opening a directory involves both downloading, verifying and decrypting the directory. The verification process is illustrated in Figure 3.10 and decryption is illustrated in Figure 3.11.

From the capability, the user either obtains the read key or the write key together with the storage index. From the server, the user receives the encrypted contents of the folder, the signature for the folder and the public key. The user verifies that the public key is correct by hashing it and matching it against a hash stored in the corresponding capability. Afterwards, the public key is used to verify the signature. If both these checks pass, the folder is decrypted with the read key and the content is obtained.

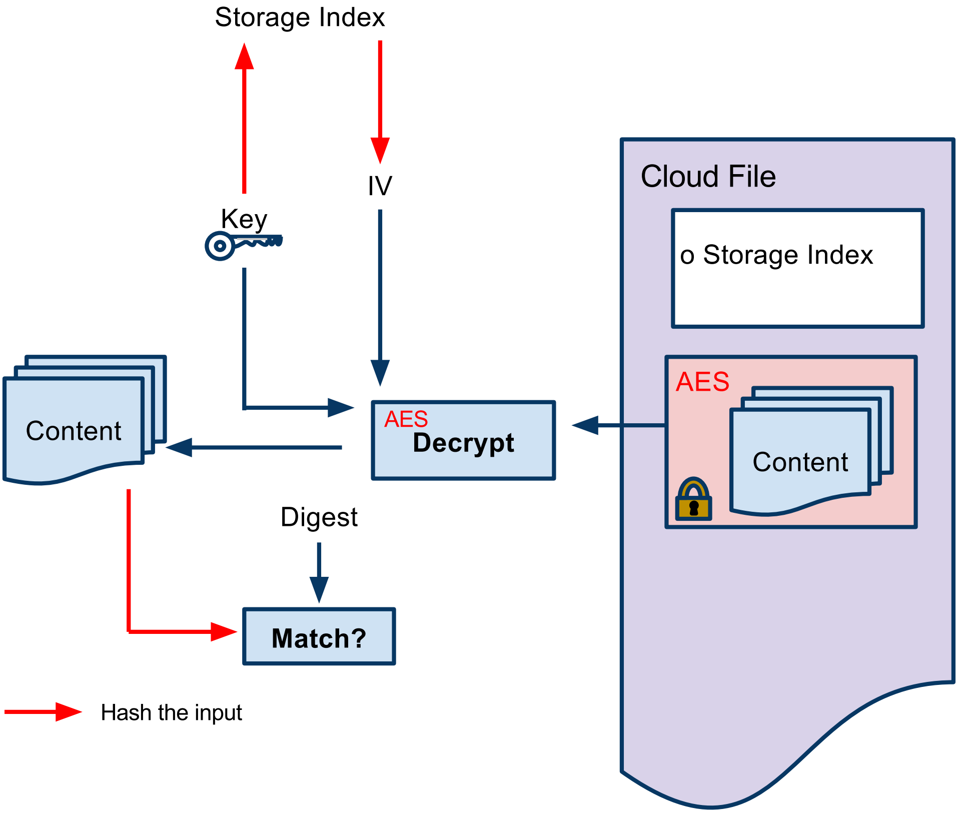


Figure 3.8: Behind the scenes: Downloading a file

### Modify Directory

In addition to read the contents of a directory, the user might want to write to it as well. The process of doing this, is similar to initially creating the first directory. The key difference, is that the user already has the write key in the form of a capability, and must use this to decrypt the encrypted signing key which resides on the server, instead of generating a new one.

A new IV is generated, and the content of the folder is encrypted, and signed by the signing key. The IV, the signature and the encrypted contents are then uploaded to the server. The write enabler is also sent alongside, which is deduced in the same way as when creating a new folder.

## 3.2.3 Recommendations for Cryptographic Primitives

For the proposed cryptographic scheme to be secure, it needs secure cryptographic primitives. More specific, it needs a symmetric cipher, a cryptographic hash function, a MAC function, a key stretching function, and a function for digital signa-

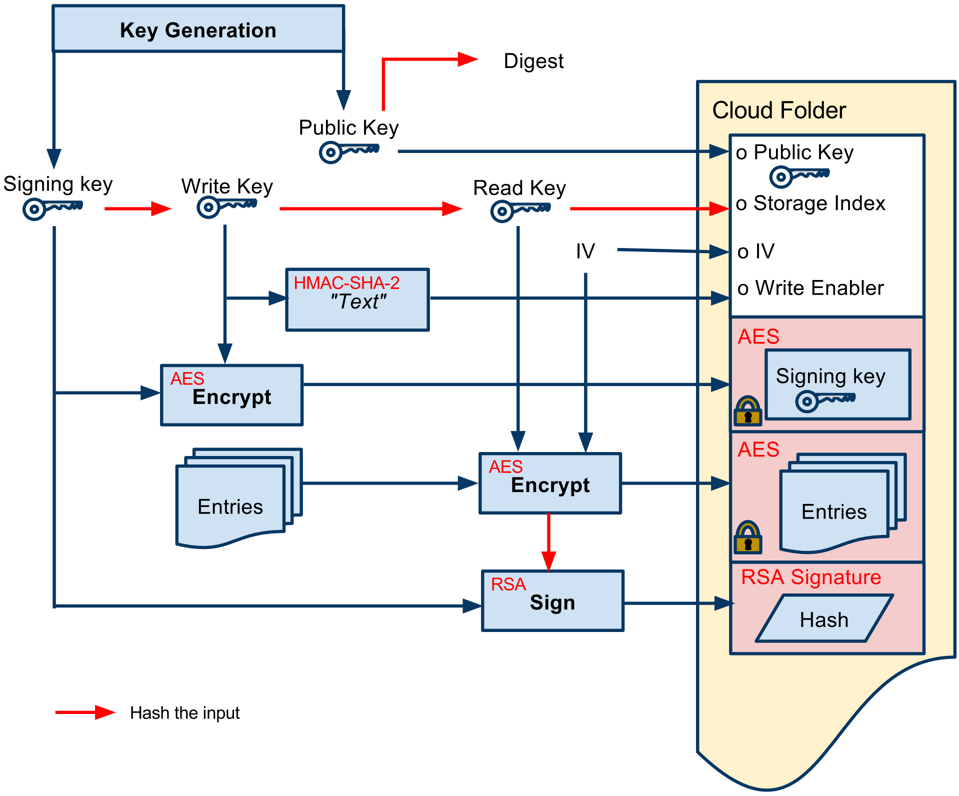


Figure 3.9: Behind the scenes: Creating a directory

tures, as observed in Figures 3.7, 3.8, 3.9, 3.10 and 3.11.

These primitives needs to be set in accordance with the security requirements established in Section 1.3. Additionally, the hard part of selecting appropriate cryptographic primitives, is trying to predict for how long the primitives will be secure. Giry [29] compares studies listing predictions on how long primitives will be secure based on different sources with different predictions.

### Symmetric Cipher

For a symmetric cipher we recommend the NIST standard AES, and a key size of 128 bit should suffice. ECRYPT II [30] has one of the more pessimistic predictions on how how secure this choice is, saying data encrypted with this key size should be secure until 2030-2040.

The gain of not choosing a larger key is a somewhat greater performance – AES-128 is 10 rounds while AES-256 is 14 rounds – and of course that the keys are smaller to store. However, for a new system there are not really any reason to not select a key size of 256 bit [14].

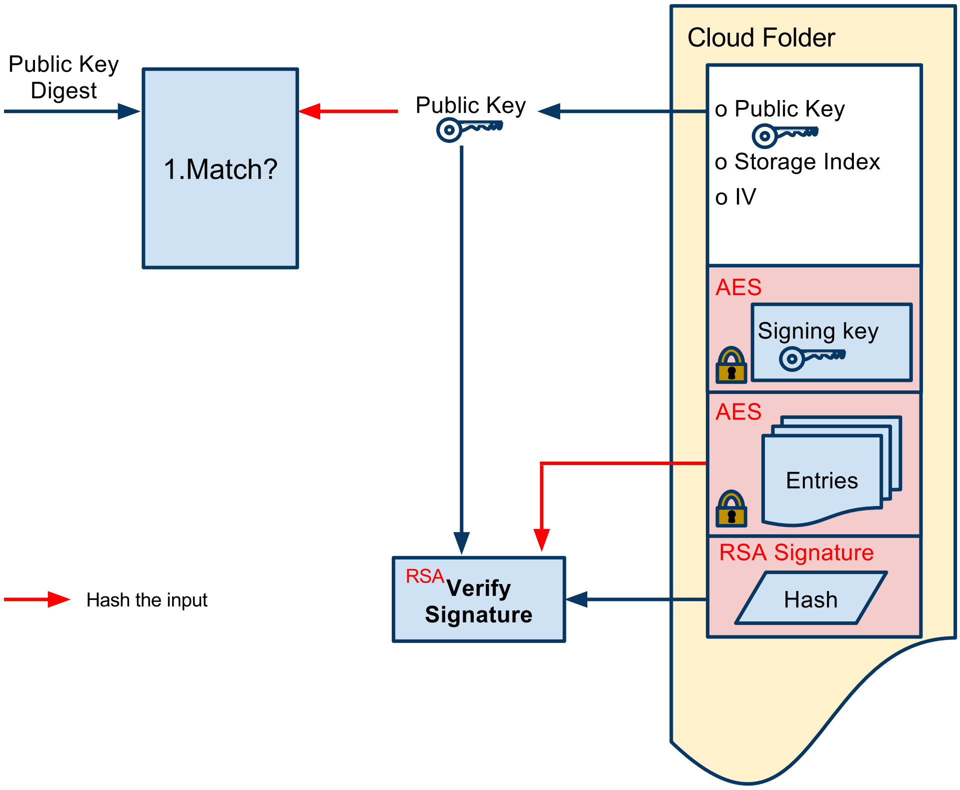


Figure 3.10: Verifying a directory

**Mode of operation** The mode of operation we recommend is CBC for the encryption of file and folder contents. This is based more on practical advice than on security considerations. Ferguson et al. [14] advices the use of either CBC or Counter (CTR) mode, where CBC mode is easier to implement correctly.

For the encryption of the private key, ECB can be used, since the key is encrypted exactly once and the data is random.

**Padding** One negative consequence of using CBC, is that it requires that the plaintext is an exact multiple of the block length, i.e. 128 bit. Since this is not always the case, a padding scheme will be required. A padding scheme does not have any security implications as long as it is reversible [14], at least not for CBC. Based on this, any available padding scheme can be used, but all clients need to use the same scheme.

### Cryptographic Hash Function

The SHA-family is the current standard for cryptographic hash functions, and from this we recommend double SHA-256. The cryptographic scheme requires the hash

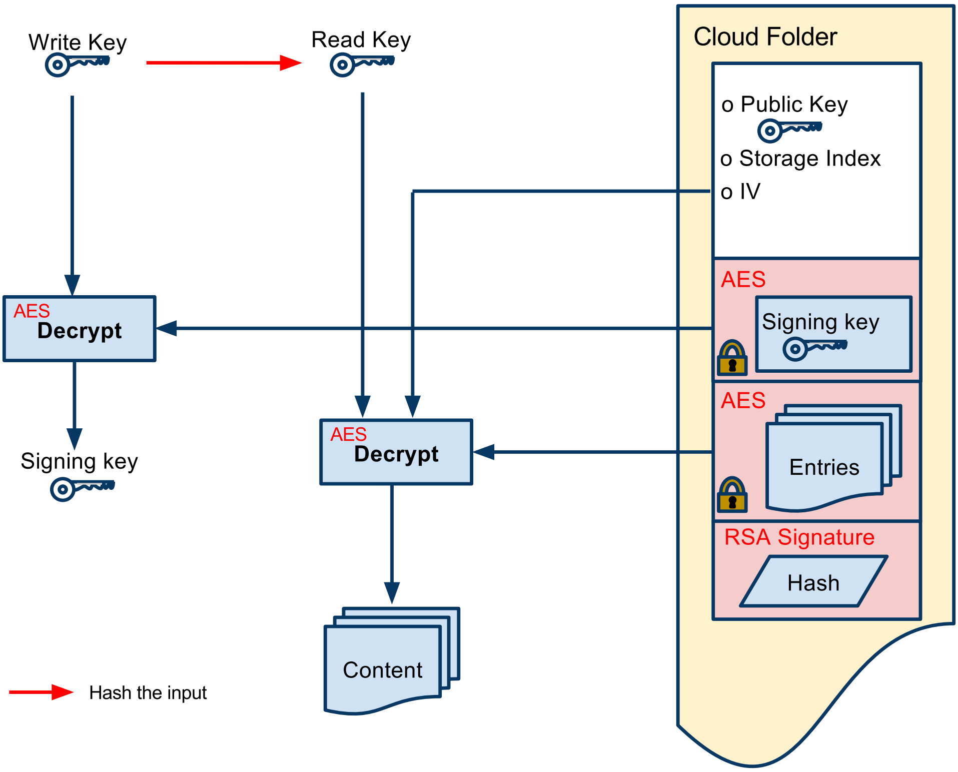


Figure 3.11: Decrypting the contents of a directory and obtaining the signing key

function to have an output of at least the size of the key used for encryption. The SHA-1 has an output of 160 bits and could have been used, but more and more attacks are discovered are not recommended for use in new systems. The use of double SHA-256 compared to single, is to prevent a length-extension attack [14].

### Signature Algorithm

For a signature scheme, the most commonly used algorithm seem to be either RSA or DSA. Both functions would work, but we recommend RSA, primarily because Tahoe-LAFS made the same choice.

There might however be a performance bonus in selecting RSA. An internet draft [31] suggests that DSA is about three times faster than RSA at signing, but RSA is about ten times faster at verifying a signature. A performance comparison from Microsoft [32] suggest that DSA is 29% faster at signing and RSA is 29% faster at verifying signatures. Verification, in the form of opening a folder, is an operation we believe most users will do significantly more than updating and creating folders.

Multiple sources [29] recommend at least 2048 bit as the key length used in RSA.

### MAC

The use of the MAC function in CSV is somewhat special. The definition states that a MAC function is used for authenticating messages. As depicted in Figure 3.9, the output of the MAC function is another key. By presenting this key, a user verifies *to the server* that he is in possession of the write key for a folder and thereby authenticated and authorized to change the folder contents. The key difference is that the MAC is used to verify that a user has write access, and not the contents of the message sent.

Because of the usage of the MAC as a simple key derivation function, the most important factor to consider when choosing a primitive, is that it should be infeasible to go from the result, back to the original key. On the basis of this, Ferguson et al. [14] recommend HMAC-SHA256.

### Password Based Key Derivation

To protect the secret that has to be stored on the terminal of the users, our recommendation is the key derivation function PBKDF2, as backed up by the NIST [27]. The iteration count for PBKDF2 is recommended to be as high as possible, while at the same time maintaining acceptable performance, while the salt length should be at least 128 bit and randomly generated.

To come up with a secure recommendation, we looked at the usage of PBKDF2 in Wi-Fi Protected Access (WPA), and what security it provides against brute force attacks. WPA is an excellent measuring unit, since the utilization of PBKDF2 is quite similar to the one in our scheme. So far, the most sensational brute force attack against WPA was published on Black Hat DC 2011 by a security researcher named Thomas Roth. He proved that anyone can crack WPA passwords with a speed of up to about 400 000 passwords per second, using multiple Amazon Elastic Compute Cloud (EC2) cluster Graphics Processing Unit (GPU) instances [33]. His findings also indicate that Amazon themselves can reach an even higher unknown brute force speed. The same researcher has also hinted that he might be able to reach 1 million keys per second on the same equipment [34].

With this in mind, we recommend at least the same amount of iterations to be used in our scheme as in WPA, i.e. 4096. This should yield acceptable performance on most devices.

**Password requirements** If we assume every password can only consist of the English letters, both upper-case and lower-case, and the numbers 0-9, there exists $62^n$ different passwords for a password of length $n$. With the stated theoretical results of 1 million passwords per second, this means that any password of length 6, 7 and 8 is cracked in about 0.66, 40 and 2527 days respectively.

These results only holds if we can guarantee that a user actually chooses a random password, which is probably not true either. A more realistic setting is that the attacker uses some dictionary, and that a user's password is not strictly random. By this rationale, it is more important to be safe than sorry, and for password guidelines we therefore recommend:

- Password length >= 9.

- Password must include at least one capital letter, one small letter and one number.

## 3.3 Server Implementation

The server, in the most basic form, has to support two operations – sending and receiving files. In addition, an extra layer is needed to support user management and access control to able to allow modification of folders.

### 3.3.1 Communication and Architectural Patterns

By definition, cloud applications are accessible over the Internet. The system we are creating, should be able to send and receive files and information from a server in the cloud. The Hypertext Transfer Protocol (HTTP) is the foundation of data communication for the World Wide Web. The protocol is well tested, will pass through most firewalls and has a multitude of available libraries in programming languages. To get a working server, we can also use any existing web server as a foundation, which will decrease total development time. Thus, HTTP was chosen as our communication protocol.

#### REST

The Web is built around an architectural style called Representational State Transfer (REST) [35, ch. 5], which is defined by four interface constraints: identification of resources, manipulation of resources through representations, self-descriptive messages, and, hypermedia as the engine of application state. In addition, REST dictates five[2] architectural constraints [35]. Our server application adheres to these constraints, or *patterns*, as follows:

**Client-server**
This server will be the server part of the client-server pattern.

**Stateless**
Since the server is just a simple key-value file store, it does not need to keep state.

**Cacheable**
The server could easily add caching, by putting each encrypted file in memory as downloaded, and e.g. using the Least-Frequently Used algorithm for choosing which items to swap out. In addition, for every update of a folder, the corresponding cache item has to be marked as invalid.

[2] And one optional, Code on demand, which is not applicable for our system.

**Layered system**

Layers are used to encapsulate, separate and hide functionality. Figure 3.12 illustrates the layers of the server application.

**Uniform interface**

The interface between clients and server(s) are given by the URI scheme in Table 3.2. A Write Enabler must also be provided together with the storage index when a folder is uploaded.

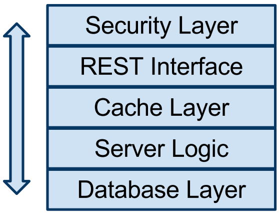


Figure 3.12: Architectural layers in the server application.

Table 3.2: The REST interface of the server application.

| URI | Description |
|---|---|
| `/put/<storage index>` | Creates or updates encrypted file |
| `/get/<storage index>` | Retrieves encrypted file |

In this context, resources are the encrypted files, and the architectural constraints of REST also matches that of our system as a whole. Thus, the server application are designed in a RESTful manner.

**Network Security**

Since we are utilizing HTTP, we can easily add an extra layer of TLS to form HTTPS. This makes it more difficult for potential attackers to intercept messages, and also provides protection against the MITM attacks. It also provides protection against eavesdropping, which would have revealed the Write Enabler for folders and enabling an attacker to delete folders. The top-most layer of Figure 3.12 thus refers to TLS.

### 3.3.2 Environment

The Python programming language in a Linux environment was chosen as development platform, together with a set of applications, interfaces and micro frameworks. The rationale for each of these follows.

**Python** Python is a high-level general-purpose programming language. It was chosen due to previous knowledge and experience by the authors, in addition to its simplicity.

**Apache** The Apache HTTP Server is a well tested and used web server. According to Netcraft [36], Apache is by far the most used web server software, and has been so since 1996. It was chosen on the basis of previous experience and its superb documentation.

**WSGI** The Python Web Server Gateway Interface (WSGI) is, as the name suggests, an interface between a web server and a Python application. It is defined in Python Enhancement Proposal (PEP) 3333[3], and specifies both sides of the interface – the *application* and the *server*. The server side is implemented in the form of an Apache module, namely `mod_wsgi`, and the application is where we put our code.

For each of the requests the server receives, a call to the application function is made with two arguments – a data structure containing the environment variables, and a callback function for which the application uses to return data to the requesting user via the server.

**Pyroutes** To adhere to the DRY principles, we chose to make use of an open source micro framework around WSGI, called Pyroutes[4]. It provides short cuts for the most frequently used functionality when developing web services, as that of Uniform Resource Locator (URL) handling and processing of submitted user data in the form of `GET` and `POST` requests.

Pyroutes did not, however, support the HTTP `PUT` request, so this was implemented and contributed back to the project.

### 3.3.3 Implementation Details

The code was structured as illustrated in Figure 3.13. The file `handler.py` provides the interface for `mod_wsgi` and the server application, and basically includes the URL scheme in `fileserver.py`. An example URL mapping is shown in Listing 3.1. The function `get_file` is registered to have the URL `/get` through the decorator provided by Pyroutes. After retrieving the file from disk, a proper HTTP response is returned, containing required headers.

The file `filesystem.py` contains the low-level file system operations, `save_file()` and `retrieve_file()`, together with a set of helper functions to manage file access checking and database operations. The folder `sql/` contains Structured Query Language (SQL) code to create necessary tables in the database, and `db.py` provides an helper function to connect to the database.

[3] `http://www.python.org/dev/peps/pep-3333/`

[4] `http://www.pyroutes.com/`

```
|-- cloudstorage
|   |-- __init__.py
|   |-- db.py
|   |-- fileserver.py
|   |-- filesystem.py
|   |-- settings.py
|   `-- sql
|       `-- write_enablers.sql
|-- handler.py
`-- tests
    `-- filesystem_tests.py
```

Figure 3.13: Server module structure

Listing 3.1: URL mapping in fileserver.py

```python
from pyroutes import route
from pyroutes.http.response import Response

from cloudstorage.filesystem import (retrieve_file,
                                     FileSystemException)

@route('/get')
def get_file(request, storage_index=None):
    if storage_index is not None:
        try:
            file_to_send, size = retrieve_file(storage_index)
        except FileSystemException, e:
            return Response(e.text, status_code=e.code)

        headers = [('Content-Type', 'application/octet-stream'),
                   ('Content-Length', str(size))]
        return Response(file_to_send, headers)

    return Response('No resource ID given.', status_code=400)
```

### Authorization functionality

The only functionality of the authorization layer implemented, is the server-side verification that a client has proper access to overwrite a folder, e.g. when a client wishes to update a folder with new contents.

When a client first uploads a new folder, it provides a *Write enabler*, which the server adds to the database along with the *Storage Index* of the folder. For every subsequent request to write to this folder, the server verifies that the provided Write enabler is equal to that in the database.

If a client tries to put a folder with a Storage Index that already exists, the server replies with an error code if the client in addition does not provide the correct Write enabler.

## 3.4 Client Implementation - Android

The proof of concept client we have implemented, is made for devices using the Android operating system, which is based on Linux. The Software Development Kit (SDK) for making Android applications, is essentially a somewhat modified version of Java.

Most devices that use the Android operating system are mobile phones or tablets, which implies that they are limited in terms of computational power and memory. The point of making the client for such a device, i.e. a *smart phone*, is the growing availability, and the flexibility these devices provide. A user carries the device everywhere, it provides network connectivity, and is almost always on.

A nice side effect of developing on a smart phone platform, is that if the software performs well on a constrained device, it will almost certainly perform just as well on any faster device.

### 3.4.1 Environment

The client was made on the Android platform and written in the Java programming language, together with a set of frameworks. The rationale for these are as follows.

**Android** The Android operating system is made by Google, and is most commonly found on mobile phones and tablets. The platform choice of Android was done based on hardware availability and familiarity with developing on the platform and the programming language.

**Java** Java is a high-level, object-oriented programming language. Applications written in Java runs in a Java Virtual Machine (JVM), which implies that a Java application can run on almost any device which has a JVM.

The "JVM" on Android is called *Dalvik*, but it is strictly not a Java Virtual Machine as the bytecode on which it operates is not Java bytecode. After the regular Java compiler has created the `.class` files, a Dalvik tool transforms them to another class file format called `.dex` [37].

**HttpComponents** Apache HttpComponents are a set of libraries for HTTP transport in Java. The part used in our client is called HttpClient. Android incorporates parts of this client in its runtime environment. The use of this library adheres to the DRY principles as defined in Section 1.5.

**JCA** Java Cryptography Architecture (JCA) is an architecture for doing cryptographic operations in Java. The architecture is based on principles of implementation independence, implementation interoperability and algorithm extensibility.

Basically what this means, is that each JVM can have different implementations of the cryptographic primitives, but the developer does not necessarily need to know which ones are available.

**ZXing Barcode Scanner** ZXing Barcode Scanner[5] is a popular Android application which can be used by other Android applications to both scan and generate barcodes. By the use of this application, we adhere to the DRY principles by not creating our own code to generate barcodes.

### 3.4.2 Architectural Patterns

**Client-Server** It should be obvious that the client we have implemented is the client part of the overall client-server pattern.

**Pipe-and-Filter** The basis of the pipe-and-filter pattern is that there exist a chain of processing elements, where the output of one element is the input of the next element. We use this for file uploads and downloads to limit the memory usage of the client, as well as to increase performance.

**Asynchronous Pattern** We use asynchronous calls to slow operations – e.g. file upload and key generation – extensively, to prevent the user interface from hanging and to deliver a smoother user experience in general.

### 3.4.3 Implementation Details

#### Structure

The source of our client is logically separated into two entities – `CSVlib` and `CSVAndroid`. `CSVlib` is a pure Java library which contain the necessary entities, cryptographic operations and communication calls required for the client. `CSVAndroid` contains primarily a graphical user interface to make use of `CSVlib` on an Android device.

#### Cryptographic Entities

All the cryptographic entities – namely folders, files and capabilities – are all part of `CSVlib`. Their relationship can be seen in Figure 3.14.

**Capabilities** Capabilities are containers for cryptographic keys and information to identify a corresponding object. They will will contain information to identify an object as either a file or a folder, and have the information to read, write or verify that object.

Capabilities are stored server side in folders in its serialized form shown in Figure 3.15. *Object Type* specifies if the capability represents a file or a directory, with values *F* or *D* respectively.

[5] `http://code.google.com/p/zxing/`

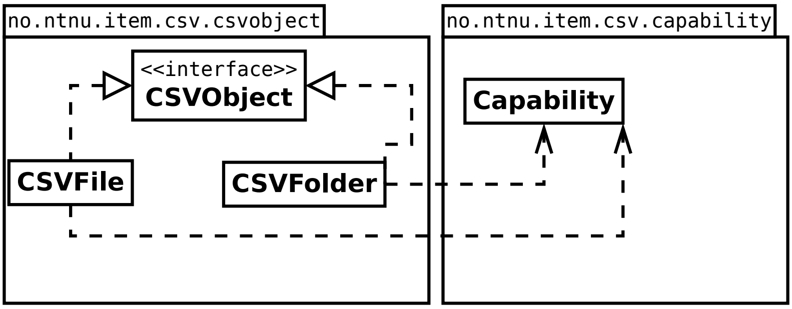


Figure 3.14: Cryptographic entities and their relations

| | | | |
|---|---|---|---|
| Object Type | Key Type | Base32(Key) | Base32(Verify) |

Figure 3.15: Serialized form of a capability

*Key Type* specifies the permissions the key will grant on the object, and can be either *RO* (Read Only) or *RW* (Read and write).

The different parts are separated by the colon character. The key and verify string are encoded in Base32, which means that the 128 bits these strings are represented by, will be replaced by an alphabet of 32 different symbols, namely A-Z and 2-7. This transformation will give some overhead in storage and transfer (26 bytes compared to 16), but makes it possible to read for a human with few mistakes or misunderstandings.

**Folders** A folder is represented by the class `CSVFolder`. A `CSVFolder` object is a collection of aliases and their corresponding capabilities. For the most part, the data stored in a folder is so small that it can easily live for as long as needed in the memory of the client.

When a folder is created or updated, the content is serialized and encrypted, before it is uploaded to the server directly from memory. The serialization for each item in a folder is simply **alias;capability**, where the capability itself is also serialized.

**Files** A file is represented by the class `CSVFile`. While a folder in general is small, a file can be of any size, even larger than the RAM on the device.

To keep the memory footprint low, we use the pipe-and-filter architectural pattern to stream data all the way from the server to the device, or vice versa, through encryption and verification.

An example of how we do this for uploads is shown in Listing 3.2.

Listing 3.2: Pipe and filter upload of a file

```
FileInputStream is = new FileInputStream("/file/path");
```

```
BufferedInputStream filebuffer = new BufferedInputStream(is);

MessageDigest md = MessageDigest.getInstance("SHA-256");
DigestInputStream digestInputStream = new DigestInputStream(
        filebuffer,md);
BufferedInputStream digBuffer = new BufferedInputStream(
        digestInputStream);

Cipher cipher = Cipher.getInstance("AES/CBC/PCKS5Padding");
// cipher is also inited with a random generated key, and an
// IV which is the digest of the key
CipherInputStream cipherInputStream = new CipherInputStream(
        digBuffer, cipher);
BufferedInputStream readBuffer = new BufferedInputStream(
        cipherInputStream);

// HttpClient uploads data by reading from readBuffer

// The digest from the file just uploaded
byte[] digest = md.digest();
```

The InputStreams are chained together, with the effect that a read from readBuffer will trigger a read trough the whole pipe. The DigestInputStream will update the state of the hash function, but is transparent in the sense that what goes into the stream will also be what comes out. The CipherInputStream on the other hand, will output an encrypted stream of the data from the file. To handle different input/output speeds, buffers are placed between each step of the stream to gain some performance.

#### Communication with Server

For HTTP transport, we utilize the Apache Software Foundations HttpComponents Client[6], also known as HttpClient. This Client offers support for authenticated requests to a server, and both upload and download through PUT and GET requests respectively.

We wrap communication with the server in two classes, Communication.java and CSVFileManager.java. Communication.java provides functionality for sending and retrieving data from our server, while CSVFileManager.java provides specific methods for sending and receiving the encrypted objects, CSVFile and CSVFolder.

### 3.4.4 Sharing

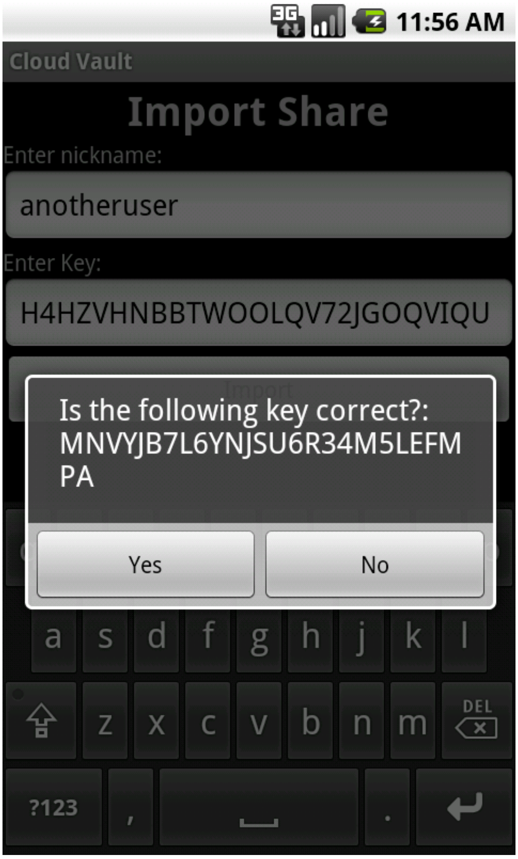
A *shared folder* is basically just a folder object which two or more people have the required encryption keys for.

The problem of creating a share with someone, is that you have to verify that you are actually sharing with the correct person. The data, or *secret*, that will have

[6] http://hc.apache.org/

to be shared, is a serialized form of the capability of the shared folder, as shown in Figure 3.15.

The client supports two methods of doing this. The most cumbersome is having to manually copy the key from one user's client to another, and afterwards verify that the key for the selected folder is correct. An example of this can be seen in Figure 3.16. This feature is also needed to support out-of-band methods for establishing a share, e.g. through the use of PGP.



Cloud Vault

Import Share

Enter nickname:

anotheruser

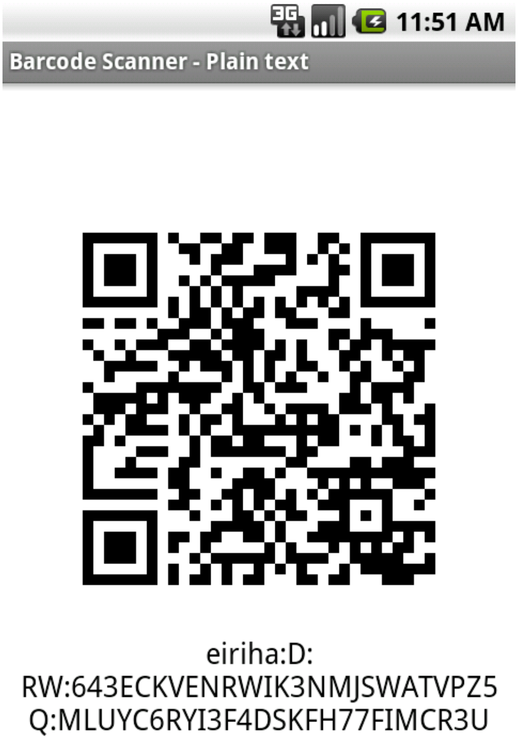
Enter Key:

H4HZVHNBBTWOOLQV72JGOQVIQU

Is the following key correct?:
MNVYJB7L6YNJSU6R34M5LEFM
PA

Figure 3.16: Establishing a share by copying the key

The other possibility is to make use of a *Quick Response (QR) code*, which is a matrix barcode that can store information. This means that one user will generate a barcode on his device, and the other user can scan that code using the camera on his device. Figure 3.17 illustrates how this code will look. The barcode contains both the key and the verification for the shared folder. Once two users have shared a folder once, that folder can be used for all future shares, which means that the two people will never have to meet and do the capability exchange again. The identity has thus been verified.



eiriha:D:
RW:643ECKVENRWIK3NMJSWATVPZ5
Q:MLUYC6RYI3F4DSKFH77FIMCR3U

Figure 3.17: Establishing a share by using barcodes

### 3.4.5 Adding a New Client

Using more clients, or different devices, is almost the exact same as sharing, and is solved in the same manner. The only difference is that the capability that needs to be transferred, is that of the root folder. It is also possible to take any other folder and use as a new root for the new client if that is the wish of the user.

### 3.4.6 Securing the Client

For a user to access his root folder, the client will have to know the capability of that folder. This is clearly too much random data for most users to remember, so the capability will have to be stored on the client. The client might be stolen or broken into by some means, and if the capability is stored in clear text, it is easily compromised. We therefore implement a local encrypted keyring which contain this capability. We use PBKDF2 together with a user defined password to derive the encryption key, and encrypt the keyring using AES-256.

### 3.4.7 User Interface

We have tried to make a user interface that is easily understandable by a novice user, both in terms of *where to click* and in terms of how we name cryptographic operations. For instance we never use the word *capability* in the client.

**Main Screen** The main screen of the application is shown in Figure 3.18. Before the user gets to this screen, he will have to unlock his local keyring with his password. If it is the first time the user starts the application, he will have to enter his online credentials, and gets a choice to either import an existing root folder, or to generate a new one. In both cases the user will have to choose a password to encrypt the root capability in to the local keyring. The most common action from the main screen would be to *Browse the vault* – in other words to see the files that the user has stored on the server.

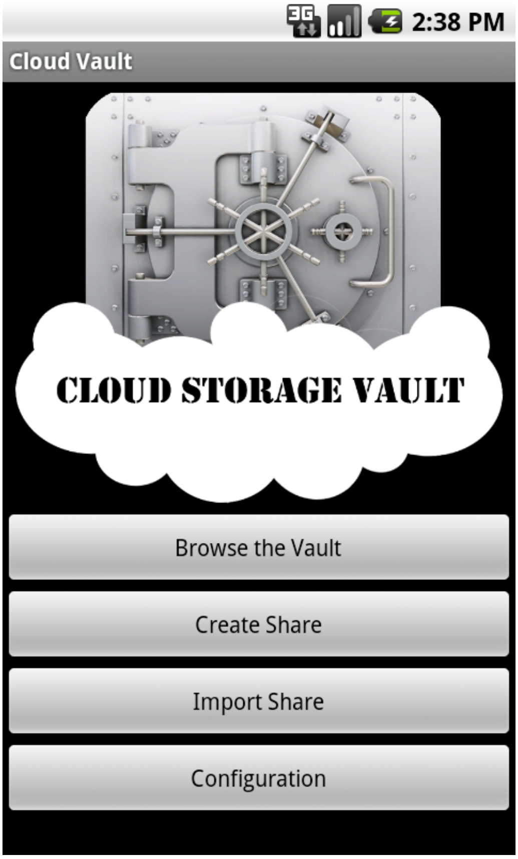


Figure 3.18: Main screen of the client application

**Browse the Cloud** The interface for browsing files stored in the cloud, is made in what we understand as a common and understandable way of interpreting users actions on the android platform, and can be seen in Figure 3.19. Tapping a file will download that file, and tapping a folder will open that folder.

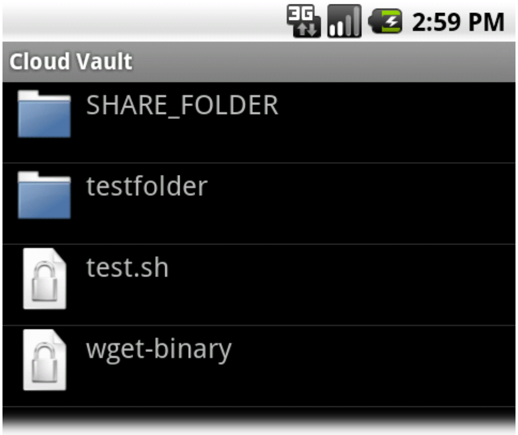
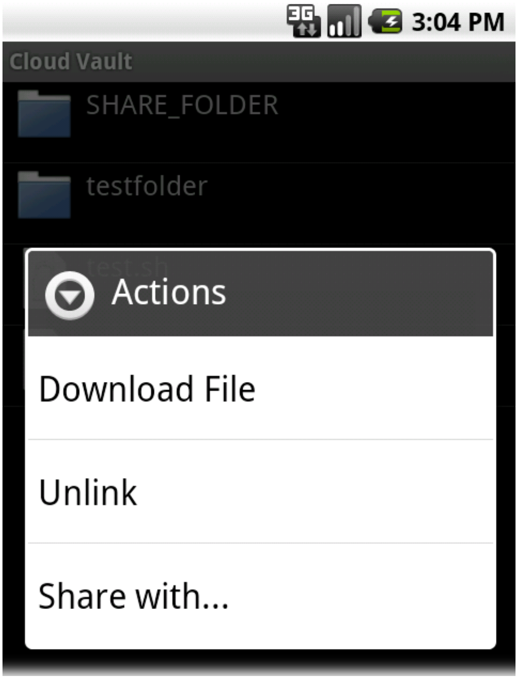


Figure 3.19: Browsing the cloud storage from the client

Selecting files to upload is treated in a similar way. To reveal this option, the user have to press the *Menu*-button. The user will then be allowed to browse his local file system for the file he wishes to upload, and tapping that file will start the upload.

A long press on a file, or a folder item, will reveal the context menu shown in Figure 3.20. The least understandable action is probably *Unlink*, which will remove a file or a folder from the parent folder. The reason why it says unlink and not delete, is that the architecture does not specify a way for deleting files. This is further discussed in Section 6.1.3.



3:04 PM

Cloud Vault

SHARE_FOLDER

testfolder

test.sh

Actions

Download File

Unlink

Share with...

Figure 3.20: Context menu showing actions available for items stored

# 4

# Experimental Procedure

This chapter describes the experimentation performed on the Cloud Storage Vault. It will start by explaining the measurements on the performance of the application. Finally, the chapter will go through the experimental procedures taken to measure the security of locally encrypted keyring.

## 4.1 Performance of the Client

We have tested the android client on two Android smartphones, an HTC Desire and an HTC Hero. We have also tested the Java libraries on a desktop computer. The specifications for HTC Desire can be seen in Table 4.1, for HTC Hero in Table 4.2, and the desktop computer in Table 4.3. The server has identical specifications as the computer used. We emphasize that the measurements are meant to be taken as indications of the performance of the scheme and implementation and not exact measurements. The Android measurements are our main objective, since that is what the client is primarily made for. The server was never considered as a possible bottleneck for the system, due to all heavy operations being performed client side.

### 4.1.1 Measured Operations

The construction used to encrypt, hash and upload *files*, is a pipeline as described in Section 3.4.3. The interesting thing to measure, is the overall speed of the pipeline, compared to a simple file upload with no *extra* operations such as hashing and encryption. To try to pinpoint the exact bottlenecks in the pipeline, we also

Table 4.1: HTC Desire Specifications

| | |
|---|---|
| Name | HTC Desire |
| CPU | Qualcomm Snapdragon QSD8250, 1 GHz |
| Memory | 576 MB RAM |
| Storage | Samsung Micro SDHC Class 2, 4 GB |
| System | Android 2.2, Linux 2.6.32.15 |

Table 4.2: HTC Hero Specifications

| | |
|---|---|
| Name | HTC Hero |
| CPU | Qualcomm MSM7200A, 528 MHz |
| Memory | 288 MB RAM |
| Storage | Micro SDHC Class 6, 4 GB |
| System | Android 2.2, Linux 2.6.29.6 |

Table 4.3: Test Computer Specifications

| | |
|---|---|
| Name | HP Compaq 8100 Elite |
| CPU | Intel Core i7 860, 4x2,80GHz |
| Memory | 4096 MB RAM |
| Storage | Hitachi HDS72105, SATA |
| System | Ubuntu Linux 10.10 64-bit, Linux 2.6.35 |
| Network | 1 GBit Ethernet, from host to server |

measure the bandwidth we get from each of the isolated operations in the pipeline on one of the Android devices, the HTC Desire.

*Folders* are relatively small in size, and the implementation does not include the use of a pipeline. All cryptographic operations are completed for the whole folder prior to sending any data. The speed of these operations are therefore measured and includes the following:

1. The average time it takes to create a folder, i.e. initial key generation
2. The time it takes to encrypt and sign a folder, with varying amount of data
3. The time it takes to verify a newly downloaded folder, with varying amount of data
4. The time it takes to serialize a folder, with varying amount of data

### 4.1.2 The Measurement Procedure

To test the bandwidth of the pipeline for files, we measure the incoming traffic to our server using the tool `nload`[1] during a file upload.

The average bandwidth are observed for a few seconds after the file upload has been started, until the file upload is nearly complete. This does not take overhead into consideration, such as the initial key generation, but for any file with a certain size, this overhead should be neglectable.

To be able to test the different folder operations in the program code, we use the Java function `System.currentTimeMillis()`, which we call before and after an operation, and calculate the difference to get the time spent. When we test operations that are dependant on the contents of a folder, we do this with each containing item being 86 bytes.

### 4.1.3 Eliminating Bottlenecks on Android Devices

On the Android devices, three possible bottlenecks that we might be able to control are identified: the *network*, the *application* and the *memory card*.

The mobile phones will normally obtain their network connection through a wireless protocol that varies naturally in throughput, e.g. Universal Mobile Telecommunications System (UMTS). These protocols work just fine, but from a measurement standpoint, we want to have a fast and stable connection. The solution was therefore to connect the Android devices to the test computer, and use the computers network through the Universal Serial Bus (USB) interface.

Another bottleneck, might be the memory card. The *class* of a memory card will identify the least sustained write speeds obtainable from the card in a fragmented state [38]. The class number $X$ represents this guarantee in $X$ MB, so a Class 2 card guarantees a speed of 2 MB/s. However, there are the possibility that the card can perform significantly better than what the class number indicates.

[1] `http://www.roland-riegel.de/nload/`

### 4.1.4 Sources of Error

The trouble with measuring performance on operations that are relatively quick, is that they are very vulnerable to *noise* from the system. The Android system comes with a lot of built-in services that runs sporadically, and hence affect the measurements. However, the small and quick operations are not necessarily fascinating to measure – the interesting behaviour to observe is how their performance is affected when the amount of data is increased. The goal of the client is to deliver a quick and smooth experience for the user.

We cannot explicitly tell what the speed of either the network nor the memory card are, and this is thus another error source. But by comparing the speed for file operations in CSV with the modified version without encryption and hashing, we should get an indication about how quick the software can be.

## 4.2 Security of the Encrypted Keyring

The security of the weakest link is measured by attacking the encrypted keyring stored by the cryptographic scheme. It is important to mention that it is the keyring pass phrase that is utilized during the attacks rather than the employed cryptographic primitives. Recall from Section 3.2.3 that the cryptographic primitives are considered secure.

The locally stored and encrypted keyring can be considered a security risk if it somehow ends up in an attacker's hands, either by a device being lost or by an intrusion in to a device. Even though the keyring is encrypted, it might be prone to a brute force or dictionary attack. If an attacker is able to decrypt the keyring, he will obtain enough information to access a user's root folder, and thus all the other files stored by the user as well.

When considering brute force attacks, it must be emphasized that it is the user's password, indirectly encrypting the keyring, that is the target weakness. Brute forcing the 256-bit AES encryption key directly will be very inefficient as it involves a key space of $2^{256}$ keys. The possibility of dictionary attacks would additionally disappear, as the encryption key is generated in a pseudo random manner.

This section describes our approach to attack the keyring.

#### Keyring Format

The format of the encrypted keyring is given in Figure 4.1. It is encrypted with 128-bit AES in ECB mode, but the key is not randomly generated. The key is given by the key strengthening function PBKDF2, but this is based on a password set by the user, which is why this key is potentially weaker than the keys for files and folders.

An attacker will also know some of the plaintext of the keyring. The serialization of a capability for a writeable folder will start with `D:RW:`, followed by 16-byte of Base32-encoded data, another `:` and then 16 more bytes of Base32-encoded data.

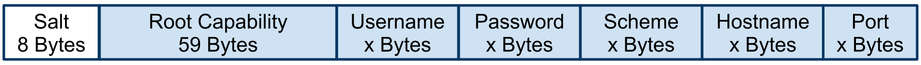


Figure 4.1: The keyring format with encrypted fields shaded in blue

### General Procedure

To perform a brute force or dictionary attack on the encrypted keyring, one must decrypt the keyring for each password guessed. The decryption involves both key derivation with PBKDF2, and decryption with 128-bit AES in ECB mode. The PBKDF2 is a function that can be used with a varying number of iterations, with the purpose of having a customizable way for users to increase key strength. Our attacks are tested on 500, 1000, 2000 and 4000 iterations.

### Implemented Attacks

We created two programs designed to crack the keyring password. The first program, named Brute Force and Dictionary Attack (BFDA), was designed for a single computer, while the second program, named Cluster Dictionary Attack (CDA), was created to perform attacks by a cluster of cooperating computers.

Both use dictionary attacks, but the firstly mentioned is also capable of a pure brute force attack. The source code and compiled `.jar` files for both programs can be found in the attached CD-ROM. Implementation details are given in Appendix A.

### Configuration

BFDA requires Java with the Java Cryptographic Extensions (JCE) and Jurisdiction Policy files which enables Java to use stronger cryptographic primitives. It also depends on Java being configured to use Bouncy Castle as its primary JCE provider. These prerequisites are required because the keyring is encrypted using Bouncy Castle, as this is used by default on the Android platform.

CDA requires almost the same configuration as BFDA, but naturally for each node in the cluster. Additionally, the cluster must be configured with Apache Hadoop.

The steps we performed to set up Java with Bouncy Castle and the Jurisdiction Policy files, are as described by Peterson [39], and the guide we used for configuring Apache Hadoop in a Cluster using Ubuntu Server, are made by Noll [40].

### Hardware Specifications

The hardware specifications for the Brute Force and Dictionary Attack are given in Table 4.4. The Cluster Dictionary Attack was executed over a cluster of Amazon EC2 instances, of type *High-CPU Extra Large Instances*. The hardware specification for a single instance, used in the cluster attack, is given in Table 4.5.

Table 4.4: Hardware Specifications for Computer Executing the BFDA

| | |
|---|---|
| Product | HP Compaq 8100 Elite SFF PC |
| CPU | Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz |
| CPU Architecture | x86_64 |
| RAM | 4GiB |
| OS | Ubuntu 10.10 (Maverick Meerkat) |
| Kernel Version | 2.6.35-28-generic |

Table 4.5: Hardware Specifications for Cluster Instances Executing the CDA

| | |
|---|---|
| Instance Type | High-CPU Extra Large Instance |
| CPU | Intel(R) Xeon(R) CPU E5410 @ 2.33GHz |
| CPU Architecture | x86_64 |
| RAM | 7GiB |
| OS | Ubuntu 10.10 (Maverick Meerkat) |
| Kernel Version | 2.6.35-24-virtual |
| I/O Performance | High (as defined by Amazon) |

### Running BFDA

The command used for executing a plain brute force attack with BFDA can be seen in Listing 4.1. The command for running a local dictionary attack is seen in Listing 4.2.

Listing 4.1: Running local brute force attack

```
$ java -jar BFDA.jar /path/to/keyring  \
    maximum_password_length number_of_threads
```

Listing 4.2: Running local dictionary attack

```
$ java -jar BFDA.jar /path/to/keyring \
    /path/to/dictionary number_of_threads
```

### Cloud Dictionary Attack with CDA

The cluster attack was carried out by 20 of the previously specified Amazon EC2 nodes. One instance was configured as both a Hadoop *master* and *slave* node, while the 19 other instances were configured as slaves. This was done to utilize as much as possible out of the available nodes, as the master node performs quite a bit less computational work than the slave nodes.

Multiple scripts are needed to initiate the distributed attack. The commands executed to start the Hadoop master and slaves, and mount up the shared, distributed file system Hadoop Distributed File System (HDFS), are shown in Listing 4.3. The final command enables the Hadoop cluster to support *MapReduce*. This is necessary as the CDA attack is implemented as a map-reduce problem. Details about Apache Hadoop, MapReduce and the implementation of CDA, are given in Appendix A.

Listing 4.3: Starting Hadoop Cluster with HDFS

```
# Start HDFS and initialize master and slave nodes
$ /path/to/Hadoop/bin/start-dfs.sh

# Start a MapReduce cluster from the master node
$ /path/to/Hadoop/bin/start-mapred.sh
```

The last requirement, before executing the attack, is to copy the desired dictionary file and keyring into the HDFS. Copying files from the master node to the HDFS is done with the command shown in Listing 4.4. The attack can then be started on the master node with the command shown in Listing 4.5.

Listing 4.4: Copying files into HDFS

```
$ /path/to/Hadoop/bin/hadoop dfs -put /path/to/file \
    /path/to/file/in/HDFS
```

Listing 4.5: Executing the CDA Attack

```
$ /path/to/Hadoop/bin/hadoop jar /path/to/CDA.jar \
    /HDFSpath/to/dictionary /HDFSpath/to/output/file \
    /HDFSpath/to/keyring number_of_slaves \
    number_of_threads_per_slave
```

# 5

# Results

In this chapter we present the quantifiable numbers retrieved when doing the experimentation described in Chapter 4. This includes performance measurements of the implemented client, and figures on the attacks presented that target the local keyring.

## 5.1 Performance of the Client

This section presents the numbers obtained from benchmarking the client, and highlights the most important results.

### 5.1.1 Files

The following sections shows the network speed obtained when uploading and downloading files. Table 5.1 displays the obtained bandwidth when using the *unmodified* client, while Table 5.2 shows the obtained bandwidth when using the same client, but with *encryption and hashing disabled*. Table 5.3 shows the speed of the individual, isolated operations in the process pipeline on the HTC Desire.

We observe that the performance of the Android devices for uploads is severely lower for uploads than for downloads, and that this is not the case for the computer client. We also notice that the speed with encryption and hashing disabled, is severely higher for all three devices. From the individual results of the HTC Desire, we identify that the encryption and decryption is the most time consuming operation.

Table 5.1: File upload/download on CSV

| **Model** | **Upload** | **Download** |
|---|---:|---:|
| Desire | 715 kB/s | 1,25 MB/s |
| Hero | 209 kB/s | 486 kB/s |
| Computer | 27,7 MB/s | 24,9 MB/s |

Table 5.2: File upload/download on CSV with encryption and hashing disabled

| **Model** | **Upload** | **Download** |
|---|---:|---:|
| Desire | 2,62 MB/s | 2,3 MB/s |
| Hero | 1,68 MB | 1,75 MB |
| Computer | ~70 MB/s | ~70 MB/s |

### 5.1.2 Folders

Table 5.4 shows the average time it takes to create an empty folder on the different devices. Table 5.5 exhibits figures on serialization of folders and Table 5.6 shows the time it takes to encrypt and sign the folder, and is visualized in Figure 5.1. These three actions are what has to be performed every time a folder is changed, while the operation of creating a blank folder is added if the content should be added to a new folder. Table 5.7 displays the time the devices used to verify an existing folder, a step which is taken by the client every time a folder is opened. Results noted as *N/A* for the HTC Hero, are operations that lead to an *Out of Memory* exception during execution.

We observe that the slowest part of folders operations are the initial key generation and more important, the serialization speed. Verification, encryption and signing is pretty fast for all devices. We also note that the performance of signing and encrypting, as well as verifying, seems to increase (in bytes/time) as the amount of data gets larger. The serialization part behaves the exact opposite where performance drops when the amount of data gets larger. We also note that our implementation struggle to handle large folders on the HTC Hero.

Table 5.3: Speed of individual operations on HTC Desire with a 4,38 MB file

| **Operation** | **Time** | **Bandwidth** |
|---|---|---:|
| Read file to memory | 1,141s | 3.84 MB/s |
| Encrypt file data | 3,761s | 1,16 MB/s |
| Decrypt file data | 3,140s | 1,40 MB/s |
| Hash data | 0,358s | 12.25 MB/s |

Table 5.4: Create a blank folder

| **Computer** | **HTC Desire** | **HTC Hero** |
|---|---:|---:|
| 81ms | 1330ms | 2060ms |

Table 5.5: Serialize the contents of a folder with n*86 bytes of data

| **n*86 bytes** | **Computer** | **HTC Desire** | **HTC Hero** |
|---|---:|---:|---:|
| 50 | <1ms | 10ms | 263ms |
| 100 | 1ms | 236ms | 376ms |
| 250 | 4ms | 943ms | 2164ms |
| 500 | 17ms | 3561ms | 8223ms |
| 750 | 42ms | 8152ms | 17230ms |
| 1000 | 71ms | 14190ms | 30397ms |
| 2500 | 487ms | 90462ms | 191312ms |
| 5000 | 1980ms | 362558ms | 756426ms |
| 7500 | 4400ms | 806255ms | N/A |

Table 5.6: Encrypt and sign the contents of a folder with n*86 bytes of data

| **n*86 bytes** | **Computer** | **HTC Desire** | **HTC Hero** |
|---|---:|---:|---:|
| 50 | 6ms | 78ms | 201ms |
| 100 | 6ms | 86ms | 272ms |
| 250 | 19ms | 33ms | 323ms |
| 500 | 5ms | 43ms | 295ms |
| 750 | 6ms | 52ms | 185ms |
| 1000 | 6ms | 69ms | 206ms |
| 2500 | 10ms | 123ms | 434ms |
| 5000 | 17ms | 317ms | 793ms |
| 7500 | 23ms | 394ms | N/A |

Table 5.7: Verify a folder with n*86 bytes of data

| **n*86 bytes** | **Computer** | **HTC Desire** | **HTC Hero** |
|---|---:|---:|---:|
| 50 | <1ms | 3ms | 10ms |
| 100 | <1ms | 3ms | 9ms |
| 250 | <1ms | 3ms | 11ms |
| 500 | <1ms | 4ms | 11ms |
| 750 | <1ms | 5ms | 13ms |
| 1000 | 1ms | 5ms | 17ms |
| 2500 | 3ms | 8ms | 26ms |
| 5000 | 4ms | 14ms | 41ms |
| 7500 | 6ms | 19ms | N/A |

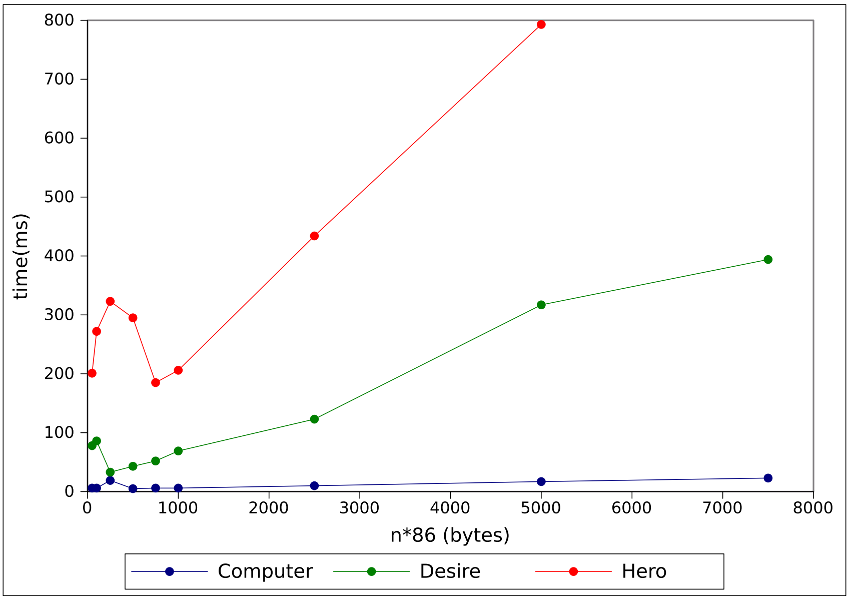


Figure 5.1: Encrypt and sign a folder

## 5.2 Brute Force Local Keyring

In Section 4.2, we tried to brute force the locally stored keyring, by implementing and executing two different programs. The first program, named Brute Force and Dictionary Attack (BFDA), was designed to run on a single computer, and supports both brute force and dictionary attacks. The second program, named Cluster Dictionary Attack (CDA), was created to run in a computational cluster. As the name implies, the CDA attack does only support dictionary attacks. The results for both programs are given in the following sections.

### 5.2.1 Brute Force and Dictionary Attack

With BFDA, we executed both a brute force and a dictionary attack to estimate the amount of passwords we could test during the course of a second.

The brute force and dictionary attacks achieved approximately the same results, although the brute force attack turned out to manage about 30 passwords more per second than the dictionary attack. A cause of this, might be due to the file reading overhead in the dictionary attack.

The results for both attacks are illustrated in Table 5.8. Results are given for different numbers of PBKDF2 iterations in passwords per second (PW/s). Figure 5.2 exhibits these numbers.

Figure 5.2 indicates that by doubling the number of iterations in PBKDF2, the efficiency of a brute force attack would decrease by half of its value, as expected.

Using 1000 iterations in PBKDF2, as recommended by RSA Laboratories [41],

Table 5.8: Speed results of running BFDA.

| PBKDF2 iterations | Brute force attack | Dictionary attack |
|---:|---:|---:|
| 500 | 3464.45 PW/s | 3451.55 PW/s |
| 1000 | 1812.01 PW/s | 1781.11 PW/s |
| 2000 | 926.53 PW/s | 883.78 PW/s |
| 4000 | 472.44 PW/s | 438.83 PW/s |

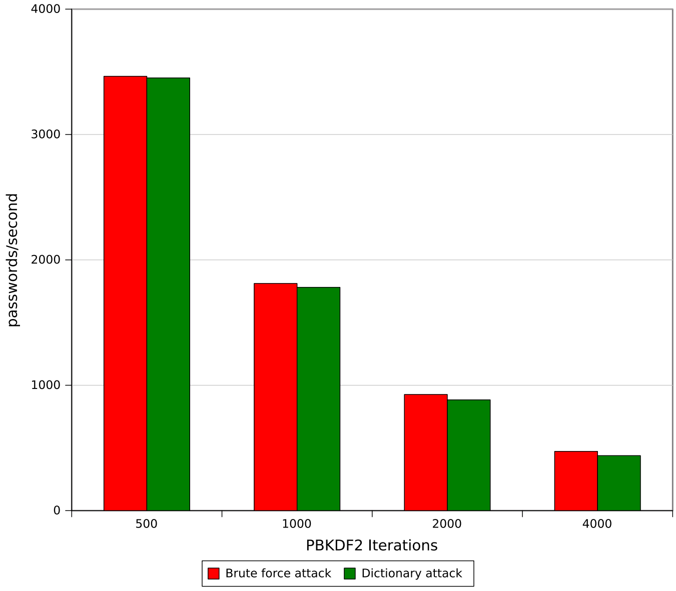


Figure 5.2: Results from running brute force and dictionary attacks against encrypted keyring.

the brute force attack was able to reach a speed of around 1800 passwords per second. With this speed, it will take, in average, 24 hours to crack a password with a length of 6 characters, consisting of only small letters.

### 5.2.2 Cluster Dictionary Attack

When running CDA, each node of the cluster achieved approximately the same results as a single computer running a dictionary attack with BFDA. We found that there was a small difference where the local computer achieved about 100 passwords per second more, compared to an instance in the cluster. This was expected, with respect to the difference in CPU power between the local computer and a single cluster instance.

With 20 nodes in the cluster, we were able to test around 8600 passwords per second, given *PBKDF2* with 4000 iterations. Running CDA in a cluster of 200 nodes, would further reach a speed of 86 000 passwords per second. Given a brute force attack with a speed of 86 000 password per second, it will take about 2496 years on average to brute force a password with the password constraints specified in Section 3.2.3. Calculations are as follows.

$$\begin{aligned}\frac{62^9 \text{ passwords}}{86000 \text{ passwords/second}} &= 157407983096.09 \text{ seconds}\\ \frac{157407983096.09}{60*60*24*365} &= 4991.37 \text{ years}\\ \frac{4991.37}{2} &= 2495.69 \text{ years in average}\end{aligned}$$

At the time of writing, renting a single *Amazon EC2 high-CPU extra large* instance costs 0.68 USD per hour. This unit price results in an hourly cost of 13.6 USD and 136 USD for clusters of 20 and 200 instances respectively.

# 6

# Discussion

## 6.1 Cryptographic Scheme

In this section, we will go through the cryptographic scheme produced and described in Section 3.2. The influence of Tahoe-LAFS is described in detail, followed by subsections scrutinizing how the scheme does or does not support various features and functionality.

### 6.1.1 Influence of Tahoe-LAFS

The general idea behind the development of the cryptographic scheme, was that we would use the scheme of Tahoe-LAFS as a basis, and simplify it where possible. One of the primary design goals was to make it easy to understand and therefore simple to accept the security features.

**Capabilities** The term *capability* is directly based on that found in Tahoe-LAFS, as it is a good descriptive name for what it is. The possession of a capability enables one to find, decrypt and verify the integrity of a file or folder.

**File Types** The concept of having two different file types, i.e. *mutable* and *immutable* files, resembles that found in Tahoe-LAFS. The only use of mutable files in the Cloud Storage Vault, is for implementing directories. By not allowing the users to generate mutable files, the scheme is significantly simplified.

**Key Generation** The process of generating keys for directories, as described in Section 3.2.2, are heavily based on the scheme of Tahoe-LAFS. By using a combination of symmetric keys and asymmetric keys, the scheme supports secure sharing of folders in both read-only and read-write modes.

### 6.1.2 Sharing

One of the major advantages of CSV over many cloud storage systems, is the possibility to share files and folders in a secure manner. If the read capability of a folder is shared, the user holding the write capability are guaranteed to be the only one capable of making valid changes to the folder.

The server denies people with only the read capability to make changes to the folder by the use of the *write enabler*. The write enabler serves the purpose of proving to the server that one holds the correct write capability. If someone with access to the file system on the server, e.g. a cloud provider employee, decides to make changes on a folder, the integrity check will warn the user of this.

**Key Distribution** The challenge of key distribution in a user friendly way is hard. To successfully authenticate a user and transfer a key, it is possible to choose from the following methods:

- Use a trusted third party
- Meet a user in person
- Use some sort of safe out of bound channel

Since one of the basic problems of this thesis is that the cloud provider is not to be trusted, it is not desirable to use a trusted third party. This implies that the only option left, is to verify another user in person, although an out of band scheme like PGP could be used.

In PGP, users can publish that they have verified other users and to which degree they trust them [12]. Other users can use this information to calculate the probability that a certain user is legit. The reason why this scheme is not implemented, is that it is fairly complex for a standard user to grasp. There are however nothing that stops a user that want to use the PGP scheme to transfer capabilities through it.

If the cryptographic scheme presented in Section 3.2 should be used in an organization, a PKI is probably the best solution. In this setting, the organization can itself be the trusted third party that enforces that all certificates granted to users are correct. A user could then simply be prompted by the name of the user he would like to share a file with, and the rest could be handled by software logic and the trust of the PKI.

### 6.1.3 Deletion of Files

To support deletion of files, various alternatives has to be assessed. Consider the following scenarios:

1. A folder is shared between two users, and one of the users has linked in the files in other directories as well. What should happen if the other user deletes the files in the shared folder?

2. A folder contains a folder which is a link to a folder “higher” in the folder tree, and thus creating a loop. By deleting the folder, should all subdirectories be deleted as well, i.e. cascading delete?

The choice of the alternatives presented shortly, are not taken by the proposed cryptographic scheme, as it is merely a practical decision.

**Creator/Access Control List (ACL)** The ACL layer on the server could store the username of the creator of the file along with each storage index, and use this to decide who should be granted access to delete files.

**Write Enabler** Similarly, the server could grant removal rights to whoever provides the write enabler, in the same manner as when updating folders. This requires that *delete enablers* is produced for immutable files.

**Link Count** The files could include a link count in its meta data, updated whenever a user links or unlinks it from a folder. When the link count reaches zero, the user notifies the server to delete it. This would have to be combined with the use of delete enablers for files.

### Loop Detection in a Directed Acyclic Graph

To remedy the problem of cascading deletion with loops, the system could utilize an algorithm for finding *strongly connected components* of a graph, before actually deleting any files.

Two components are strongly connected if there exist a path from $A$ to $B$, and from $B$ to $A$, as depicted in Figure 6.1 where a directory is linked in at a higher level in the directory tree. The dashed line represents the actual path in the graph. If the folder $X$ is requested to be deleted, the $Y$ directory will also be removed, and this is might not the behaviour the user expected.

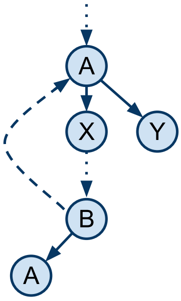


Figure 6.1: Theoretical cycle in the directory graph

*Tarjan's Algorithm* [42] is an example of algorithm that can be implemented to do this kind of check, and is basically a depth first search, with a stack containing visited nodes.

### 6.1.4 Verification of Files

The verification scheme for immutable files is suboptimal in every way a user wants to use an ordinary file. The problem is that the user will have to download the entire file, before he can verify that the file is what it is supposed to be. If it turns out that the file does not pass this check, and the error was in the very first byte, the user has wasted time and bandwidth downloading a lot of useless data. The same scenario applies if the user only wants to look at a small part of a file, e.g. the middle section of a movie.

A possible solution to support this kind of verification, would be to build a *hash tree* of the whole file, as shown in Figure 6.2. With a hash tree, smaller parts of the file can be verified, and errors can be detected earlier. The hash tree could be stored encrypted on the server together with the encrypted file, and the capability could hold the root hash of the tree. The downside to this approach, is that performance would be affected due to the substantial amount of hash operations required to compute the hash tree. This idea of a hash tree is implemented in Tahoe-LAFS [5].

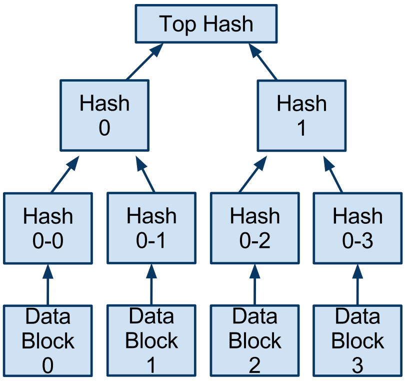


Figure 6.2: Hash tree of a file

### 6.1.5 Version Control System

Some of the currently available cloud storage systems, as Wuala and Dropbox mentioned in Section 2.6, support a kind of *version control* of the files stored. This means that the data lost during a modification of a file are saved alongside the current file, so that the user has enough information to restore the file to a previous state. In practice, this often mean to store multiple versions of the same file, or the difference between two versions. This could be implemented in CSV by one of the following methods.

**Extension to Folders** By adding a nested list to the directory structure exemplified in Section 3.4.3, we can link previous versions of a file in the serialization form:

**alias;capability;capability;capability**

To make this feasible and user friendly, a time stamp would have to be added to the capability. This would increase the size of the folder.

**Type of Mutable File** A new form of mutable file could be implemented in practically the same form as a directory, containing timestamps instead of aliases, and pointing to corresponding storage indexes. With this procedure, it should be easy to configure for the user which files and folders that should be under version control.

### 6.1.6 Deduplication

The term *data deduplication* implies not having to store redundant data. By implementing a deduplication scheme, multiple advantages can arise in the sense of a storage system. For the service provider, this could mean cost savings in the form of not having to store the same file twice, but still claim money from users for the given storage. For the users of the service, this could mean better network utilization, as uploading a big file that already exist on the server would take no time at all.

However, there are practical disadvantages and great privacy concerns related to deduplication, in addition to some solutions that address these issues. The CSV does not utilize deduplication, as the scheme simply does not support it. Since all encryption keys are randomly selected, neither the server or the client is able to detect whether a file already exists in the system.

**Practical Disadvantages** Deduplication relies on the use of hash functions to check whether a file or a block of data exist on the storage node. In general, as long as you hash something larger than the length of the digest, there is the potential for a collision, and hence data corruption.

Further on, the additional hash operations and queries to the server pose extra computational overhead, which increase if the deduplication is on block level.

**Privacy** Implicitly, using deduplication globally – i.e. for the whole system, for all users – gives any user of the system the possibility to check a file for existence. This could for instance be utilized by anti piracy companies working for the film industry, by first uploading a file known to be illegally spread over the Internet, and then see if it takes next to nothing time to upload it to the server. If this is the case, they can file a petition to get the company hosting the server to yield the contact information for the users in question.

**Customizable Deduplication** Tahoe-LAFS tries to provide a solution that has all the advantages and none of the issues related to deduplication, by *convergent encryption with an added secret* [5].

Convergent encryption implies using the hash of the plaintext as the key to the symmetric encryption algorithm, i.e. the same plaintext will always yield the same ciphertext, making it relatively easy to implement deduplication.

Tahoe-LAFS adds a per-client secret to the hashing procedure, as depicted in Figure 6.3, before using the result as a key to encrypt the file. This enables per-client deduplication, or per-group deduplication if the user shares the secret with other users. Since the storage index is based on the encryption key, and the plaintext will always lead to the same encryption key, the client can check for file existence on the server and hinder duplication.

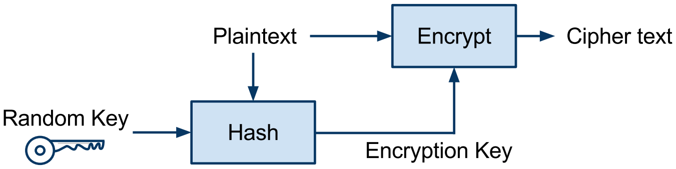


Figure 6.3: Tahoe-LAFS deduplication scheme

### 6.1.7 Supporting Multiple Cryptographic Primitives

The main problem with selecting cryptographic primitives for our scheme, is that one can not be sure how long they will remain secure. A recommended approach is to build systems that can easily change the cryptographic primitives, in case of discoveries enabling an attacker to break a primitive [14]. The main problem with this approach is that if for instance AES is broken, and the cloud provider keeps backups of all user data on their server, that information is compromised with regard to the provider.

A change of a cryptographic primitive in our scheme would lead to backward compatibility being broken, i.e. one would be unable to obtain files or folders stored before the change. This could be remedied by a meta-block for each file and folder, containing the utilized cryptographic primitives. In addition, the capability would have to contain which cryptographic hash function that should be used, to be able to deduce the storage index from the key.

## 6.2 Implementation

This section will discuss and deal with the different choices taken to fulfil the implementation of the cryptographic scheme. We will start by discussing the use case we chose to implement. Further on, we will analyse our choice of key distribution scheme and look at alternative solutions.

### 6.2.1 Choice of Use Case

There are mainly two types of user groups interested in a cloud storage solution – personal or enterprise. The proof of concept implementation is more aimed against the personal use case, than the enterprise market.

In an enterprise, the use of a TTP are often already in place, usually in the form of an IT department. An additional fundamental difference, is that it would usually be more defined who the users are supposed to share files with, and what they should have access to. An organization might not approve of a model where all users can forward read rights to other users.

### 6.2.2 Key Distribution

### 6.2.3 Deviations in the Choice of Cryptographic Primitives

The implementation of the proof of concept client deviates from the recommendation given in Section 3.2.3. This occurred as a result of new discoveries that were made while analysing the system. All unfortunate deviations are corrected with a patch file submitted along with the code attachment, described in Appendix B. The following deviations exist.

**RSA Key Length** The key length chosen for the RSA cipher is 1024 bits, in contrast to the argued 2048 bits. It is therefore important to notice that all measurements, presented in Chapter 4 and 5, are based on an RSA key length of 1024 bits.

**AES Key Length** The key length used in our implementation is 128 bit for AES. The reason to prefer a shorter key length in CSV is to make manual key exchange less cumbersome. 256 bits are however preferred.

### 6.2.4 Simplifying the Server

The implemented proof of concept server generally performs quite well, and was never identified as a bottleneck during the performance measurements. However, by making the server-side contain less logic, CSV could be used against for instance the Amazon S3 service. This would more easily enable users that are not comfortable with setting up a web server, to set up the whole system from scratch. Another possibility is that the cloud provider could offer the CSV as SaaS.

By redesigning the scheme to use a *dumb file store*, the ACL functionality will be lost. However, a service like Amazon S3 do provide some access control, and in the scenario of a *friend net*, this could actually be considered as a better solution. Luckily, the cryptographic architecture presented in Section 3.2, are loosely coupled with the ACL layer, so extending the proof of concept software to use a dumb file store should not pose any great concerns.

## 6.3 Performance

The measurements performed on our client reveal that it is not as fast as desirable. For files, a perfect client would render the network or the SD card as the bottleneck, but our measurements state that this is not the case.

Because of the cryptographic operations, the download and upload speed for a given file is decreased by around 50-80% on all three devices. The results for the individual operations performed on HTC Desire shown in Table 5.3, indicates that encryption and decryption are the slowest elements in the process pipeline, and responsible for most of the performance drop.

**Limitations of Libraries** Another thing to note, is that the implementations of the `InputStream` and `OutputStream` objects in Java and the Apache HttpComponents seems to read and write the requested number of bytes stream by stream. This implies that the implementation will never reach the speed achieved by the slowest component in the pipeline.

**Multiple CPU Cores** Branching the operations in the pipeline to different threads should make it possible to gain better performance, especially in an environment which spots multiple processor cores. Multiple CPU cores is standard on modern computers, and there is no reason to suspect otherwise than that also smart phones will be equipped with this in the nearest future. In addition, dedicated cryptographic co-processors may relieve the CPU from additional work.

**Folders** The performance of the folder operations in the implementation is quite good, if we look at the cryptographic functionality. We identify the slowest operation to be the generation of new cryptographic keys. For a user of the Android client, this should not be a problem, since the folder creation is done as an asynchronous task while the user enters the name of the new folder. Another solution is to have the application maintain a pool of keys before they are requested by the user. In this way a key can almost always be ready for the user when he needs it.

The other cryptographic operations on folders completes quite quickly, but the performance for serializing the folder contents is too low when folders contain a large number of items. A folder with thousands of children is not something we expect an ordinary user of our Android client to have, but if we change the scenario to backing up an entire computer, the serialization part might become a problem.

The slow serialization operation can partially be blamed on our own implementation. The folder items are stored in a `HashMap`, which is serialized to a string to make it ready for encryption. Ideally, this step is not necessary, given that the folder contents exists in its correct form in memory. This might not be possible due to reasons such as decreased performance when manipulating the folder content. But we believe it is possible to create a faster implementation of the currently utilized serialization algorithm.

**Cache** Since opening folders is a relatively expensive procedure, given that it has to be downloaded from the server, decrypted and verified, we keep the currently opened folder and its parents in a stack. This can be seen on as a kind of cache, and could be extended to include all folders opened in the active session.

## 6.4 Security

The secure storage system described in this thesis is based on the idea of hiring storage from an untrusted provider. Thus, the security of the application should primarily reflect that. With this as a starting point, we can assume that all data stored on the server is obtainable by the provider.

This section will discuss the general security provided by the Cloud Storage Vault, and is further divided into security of the cryptographic scheme and security of the user client.

### 6.4.1 Security of the Cryptographic Scheme

The proposed cryptographic scheme is subject to certain weaknesses that currently, or in the future, may be utilized by an attacker. The following subsections will discuss possible weaknesses and their consequences.

#### Information leakage

The storage provider can access all encrypted data on the server, and also know whether a stored data object corresponds to a file or a folder. This is revealed by the difference in header content and size between folder and file objects. Folders are specially recognizable by the server, as their write enabler is revealed during upload procedure.

Additionally, a user's root folder is known, since it is the first object accessed in a newly created session. It is also possible for the provider to reveal the structure of the directory tree of a user. The series of file and folder requests from a user enables the provider to learn which files are contained in which folders.

Considering the findings in various research, information can also be leaked to other potential attackers besides the untrusted cloud provider. External attackers can locate the server in the cloud, used by the application, and further set up a VM on the same server [13]. In a worst case scenario the attacker gets access to the same amount of information as the cloud provider, which should not be dangerous.

#### Content Disclosure

The confidentiality of both files and folders relies primarily on the symmetric cipher used to encrypt the data. But for folders one can obtain this key if one manages to obtain the asymmetric private key belonging to the folder, as seen in Figure 3.9. In other words, files and folders are attackable both trough the asymmetric and the symmetric cipher used.

Disclosing folder content through the asymmetric cipher may be avoided by changing the usage of the private key. Instead of hashing the private key to derive the write key of a folder, the write key could rather be generated from a random key generator. In this way, even though the asymmetric private key is revealed, the symmetric keys used to provide confidentiality are not retrieved. However, revealing the private key would still breach the integrity of the cryptographic scheme.

If the confidentiality of a folder is breached, it will additionally enable the attacker to decrypt all of the corresponding subfolders and files. In contrast, cracking the confidentiality of a file will only compromise that specific file. It is also important to note that if the confidentiality of a user's root folder is breached, all files, belonging to that user, are effectively compromised.

#### The Weakest Link

Considering the potential weaknesses discussed above, the weakest link in the secure cloud storage scheme presented, is potentially the locally encrypted keyring. To provide usability, users are given the possibility of choosing their own password, to indirectly encrypt their keyring. However, it is fairly well known that users can not be trusted to generate and use long, secure passwords.

As mentioned earlier, even with the recommended password constraints, the keyring is still vulnerable to dictionary attacks. However, disclosing the keyring is only possible if the keyring is accessible to an attacker. Getting access to a locally stored keyring is defined as an *active attack*, and is further discussed in the next section.

### 6.4.2 Client Security

If a user physically looses his device, and the CSV software is running, anyone who finds the device can get access to the user's private files and folders. This is because the client software will have to keep the root capability from the keyring in memory, and will potentially have copies of the files on the device. The same applies if the device is broken into, e.g. through a vulnerability in the operating system.

#### Downloaded Files

An attacker who gains access to the device running CSV will have be able to read files downloaded. We can avoid writing downloaded files to the persistent storage medium. This can be implemented using a temporary file system that only persists in the memory of the client, e.g. *ramfs*. This is a file system for clients running Unix-like operating systems. However, by utilizing this method, will limit the possible size of the downloaded files to the size of the available memory.

In case limited file size is not an acceptable solution, it is possible to use a temporary file system that supports swapping, e.g. *tmpfs*. This will, however, potentially keep parts of a large file on the persistent storage medium, revealing information about a downloaded file. A practical feature would be to store the encrypted files on the persistent storage medium, and only put them in the temporary file system when needed.

### Disclosing the Root Capability

The confidentiality of the root capability is held by an encrypted keyring that is stored client-side. The keyring is encrypted with a key derived from a password provided by the user. Unfortunately, using a password to encrypt the keyring, makes the keyring a suitable target for brute force and dictionary attacks. The vulnerability of the keyring against brute force and dictionary attacks was questioned in Section 4.2.

**Brute Forcing the Keyring** From the results in Section 5.2 and Figure 5.2, we noticed that the choice of iterations in PBKDF2 were conclusive to the efficiency of a brute force attack against the local keyring. By doubling the number of iterations in PBKDF2, the efficiency of a brute force attack would decrease by a corresponding 50%.

It is important to notice that both of the CDA and the BFDA programs were written in Java. With this in mind, it is reasonable to believe that general performance can be improved by using a lower-layer programming language, such as C, to implement the attacks. Another way to increase performance is to design both attacks to run on GPUs rather than CPUs.

When following the recommendations for PBKDF2 and password constraints given in Section 3.2.3, it is infeasible to perform a brute force attack against the local keyring. However, even though the keyring is secured against a plain brute force attack, it is still vulnerable to dictionary attacks. To completely prevent such attacks, one would have to make sure that the encrypted keyring is never disclosed.

**Active Attacks** While the encrypted key ring may resist some dictionary and brute force attacks, other problems arise when facing an active attacker. If the attacker is able to read the memory of the device, he can also read the root capability if this is unlocked by the user.

A mechanism that decrease the risk of this happening, is to have the software time out after a specified interval. At the timeout the software can overwrite the sensitive information in memory.

Another scenario would involve the attacker installing a *keylogger* on the compromised client. This type of software can further detect the keyring password when the user logs in to the application.

# 7

# Conclusion and Future Work

The main purpose of this thesis was to create and implement a solution to a secure cloud storage service. The scheme was to provide what we defined in Section 1.3 as a secure storage system.

Key elements of the proposed cryptographic scheme is based on the secure and relatively scrutinized file system Tahoe-LAFS. With this in mind, and the use of cryptographic primitives recommended in Section 3.2.3, the proposed cryptographic architecture is believed to offer a secure storage system.

Findings in Section 5.2 showed that the presumably weakest link of the system, the encrypted keyring, is able to withstand brute force attacks. However, to acquire the keyring, an active attack on the client device is required. In addition, measuring performance on constrained devices proved that the trade-off between the levels of usability and security did not undermine the desired strength of the underlying cryptographic primitives.

In addition to confidentiality, the proposed cryptographic scheme support secure sharing of the files stored on the server. A solution to the problem of key distribution – to be able to create a *trust* between users – were implemented using barcode scanning, and proved to sustain existing security. The scheme does not require a specific key distribution method, but it should be possible to implement any solution on top.

When developing a secure cloud file system, one of the greatest challenges turned out to be the choice and implementation of cryptographic primitives. Good references and external recommendations can help, but only to a certain degree. Key distribution is also a general problem that will always be hard to accomplish, if a trusted third party is not to be included. Yet another challenge experienced, is that all additional features desired for a secure cloud file system, are hard to implement without compromising security. Deduplication is an example of this.

In this thesis, we have shown that it is possible to create and implement a fundamental cryptographic scheme for a secure cloud storage system, supporting sharing. Our main contribution is a simple, yet secure, cryptographic scheme for securely storing and sharing data in the cloud. We have also implemented an open source, proof of concept server and client for Android devices, that have further been measured in performance and evaluated in security.

While our scheme solves some of the basic challenges of security within cloud storage, there are several features that could improve the scheme. The most important being a better approach to verifying files, to e.g. allow streaming, and the possibility of deleting files. For our implementation it would be interesting to see a modified version which could work directly with an existing *dumb* cloud storage solution, such as Amazon S3.

# Bibliography

[1] Jeremy Geelan. Twenty-One Experts Define Cloud Computing. `http://cloudcomputing.sys-con.com/node/612375`, January 2009. Retrieved 05.13.11.

[2] P. Mell and T. Grance. The NIST Definition of Cloud Computing (Draft). Technical report, NIST, 2011. From `http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf`.

[3] Google history. `http://www.google.com/corporate/history.html`. Retrieved 05.15.11.

[4] Windows Live - All services. `http://home.live.com/allservices/`. Retrieved 05.15.11.

[5] Zooko Wilcox-O'Hearn and Brian Warner. Tahoe - The Least-Authority Filesystem, 2008. From `http://tahoe-lafs.org/~zooko/lafs.pdf`.

[6] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. Towards trusted cloud computing. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association. From `http://www.mpi-sws.org/~gummadi/papers/trusted_cloud.pdf`.

[7] W. Itani, A. Kayssi, and A. Chehab. Privacy as a Service: Privacy-Aware Data Storage and Processing in Cloud Computing Architectures. Technical report, American University of Beirut, 2009. From `http://www.csd.uwo.ca/courses/CS9842/PaperReviews/PrivacyAsAService.pdf`.

[8] S. Pearson, Y. Shen, and M. Mowbray. A Privacy Manager for Cloud Computing. Technical report, HP Labs, 2009. From `http://www.hpl.hp.com/techreports/2009/HPL-2009-156.html`.

[9] Seny Kamaraand and Kristin Lauter. Cryptographic cloud storage. In *Financial Cryptography and Data Security*, Lecture Notes in Computer Science. Microsoft Research, Springer Berlin / Heidelberg, 2010. From `http://dx.doi.org/10.1007/978-3-642-14992-4_13`.

[10] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[11] Andrew Hunt and David Thomas. *The Pragmatic Programmer: from journeyman to master.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[12] William Stallings. *Cryptography and Network Security – Principles and Practices.* Pearson Education Inc., fourth edition, 2006.

[13] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, New York, NY, USA, 2009. ACM. From `http://cseweb.ucsd.edu/~hovav/dist/cloudsec.pdf`.

[14] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications.* Wiley Publishing Inc., 2010.

[15] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. URL `http://www.ietf.org/rfc/rfc2104.txt`. Updated by RFC 6151.

[16] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), September 2000. URL `http://www.ietf.org/rfc/rfc2898.txt`.

[17] A. C. Yao. Protocols for Secure Computations. Technical report, University of California Berkeley, 1982. From `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.7844&rep=rep1&type=pdf`.

[18] C. Gentry. Fully Homomorphic Encryption Using Ideal Lattices. Technical report, Stanford University and IBM Watson, 2009. From `http://www.math.uni-bielefeld.de/~mitrofan/p169-gentry.pdf`.

[19] A. Narayanan and V. Shmatikov. Obfuscated Databases and Group Privacy. Technical report, University of Texas, Austin, 2005. From `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.7808&rep=rep1&type=pdf`.

[20] Dropbox. Dropbox Reveals Tremendous Growth With Over 200 Million Files Saved Daily by More Than 25 Million People. `http://www.dropbox.com/press/release`, . Retrieved 04.27.11.

[21] Dropbox. How secure is Dropbox? `http://www.dropbox.com/help/27`, . Retrieved 04.27.11.

[22] Miguel de Icaza. Dropbox Lack of Security. `http://tirania.org/blog/archive/2011/Apr-19.html`. Retrieved 04.27.11.

[23] Barry A. Cipra. The Ubiquitous Reed-Solomon Codes. *Society for Industrial and Applied Mathematics (SIAM) News*, 26-1, January 1993. From `http://www.eccpage.com/reed_solomon_codes.html`.

[24] Dominik Grolimund, Luzius Meisser, Stefan Schmid, and Roger Wattenhofer. Cryptree: A Folder Tree Structure for Cryptographic File Systems. In *SRDS'06*, pages 189–198, ETH Zurich, CH-8092 Zurich, 2006. From http://dcg.ethz.ch/publications/srds06.pdf.

[25] Dominik Grolimund. Google Tech Talk: Wuala - A Distributed File System. http://www.youtube.com/watch?v=3xKZ4KGkQY8, 2007. Seen 31.05.11.

[26] Mark Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. Technical report, Combex Inc., UC Berkley, Johns Hopkins University, 2003. From http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.6.3660&rep=rep1&type=pdf.

[27] Meltem Sönmez Turan, Elaine Barker, William Burr, and Lily Chen. Recommendation for Password-Based Key Derivation (Draft). Technical report, NIST, June 2010. From http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.169.1626&rep=rep1&type=pdf.

[28] John Kelsey, Bruce Schneier, Chris Hall, and David Wagner. Secure Applications of Low-Entropy Keys. In *Information Security Workshop (ISW'97)*, September 1997. From http://www.schneier.com/paper-low-entropy.pdf.

[29] Damien Giry. Cryptographic Key Length Recommendation. http://www.keylength.com/en/compare/, 2011. Retrieved 24.05.11.

[30] European Network of Excellence in Cryptology II. Ecrypt II Yearly Report on Algorithms and Keysizes. Technical report, ECRYPT II, March 2010. From http://www.ecrypt.eu.org/documents/D.SPA.13.pdf.

[31] P. Hoffman. Dsa with sha-2 for dnssec draft. Technical report, IETF, July 2009. From http://tools.ietf.org/pdf/draft-hoffman-dnssec-dsa-sha2-00.pdf.

[32] Microsoft. Performance Comparison: Security Design Choices. http://msdn.microsoft.com/en-us/library/ms978415.aspx, October 2002. Retrieved 23.05.11.

[33] Thomas Roth. Breaking encryptions using GPU accelerated cloud instances. https://media.blackhat.com/bh-dc-11/Roth/BlackHat_DC_2011_Roth_Breaking_encryptions-wp.pdf, 2011. Retrieved 05.10.11.

[34] Thomas Roth. Upcomming black hat talk. http://stacksmashing.net/2011/01/12/upcoming-black-hat-talk/, january 2011. Retrieved 29.05.11.

[35] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[36] Netcraft. January 2011 Web Server Survey. http://news.netcraft.com/archives/2011/01/12/january-2011-web-server-survey-4.html. Retrieved 04.27.11.

[37] DalvikVM.com. Brief overview of the Dalvik virtual machine and its insights. http://www.dalvikvm.com/, 2008. Retrieved 05.21.11.

[38] SD Specifications Part 1 Physical Layer Simplified Specification. http://www.sdcard.org/developers/tech/sdcard/pls/simplified_specs/Part_1_Physical_Layer_Simplified_Specification_Ver3.01_Final_100518.pdf, 2010. Retrieved 05.15.11.

[39] Zachary Peterson. Installing Policy Files and Bouncy Castle Provider. http://znjp.com/mcdaniel/BC.html, 2008. Retrieved 05.04.11.

[40] Michael G. Noll. Running Hadoop On Ubuntu Linux (Multi-Node Cluster). http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/, 2007. Retrieved 05.04.11.

[41] RSA Laboratories. PKCS #5 v2.1: Password-Based Cryptography Standard. http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf, 1999. Retrieved 05.10.11.

[42] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, Vol. 1, 1971. From http://epubs.siam.org/sicomp/resource/1/smjcat/v1/i2/p146_s1.

[43] Matthias Gärtner. PBKDF2 Java implementation. http://narenst.wordpress.com/tag/pbkdf2/, 2007. Retrieved 05.05.11.

[44] Apache Hadoop. Project Description. http://wiki.apache.org/hadoop/ProjectDescription, 2009. Retrieved 05.06.11.

[45] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters, 2004. From http://labs.google.com/papers/mapreduce-osdi04.pdf.

# Appendices

# A

# BFDA and CDA Implementations

This chapter contains details about the implementations of BFDA and CDA.

## A.1 Brute Force and Dictionary Attack

*BFDA* includes a plain brute force attack and a dictionary attack against a users encrypted keyring. Details about the implementation follows.

### A.1.1 Implementation Details

BFDA is written in Java and utilize the `javax.crypto` library with Bouncy Castle version 1.34 as JCE provider to decrypt the encrypted keyring. The Bouncy Castle JCE provider seems to be necessary as Android 2.2 uses it by default to encrypt the keyring. To enable *PBKDF2* before decryption, BFDA utilize a *PBKDF2* Java library [43].

The program is divided into two functions, named `bruteForceAttack` and `dictionaryAttack` that correspondingly executes a brute force and dictionary attack. The `bruteForceAttack` function is shown in Listing A.1.

Listing A.1: bruteForceAttack function

```java
public void bruteForceAttack(String[] input) {
  current_word = new char[1];
  words = new Stack<String>();
  THREADS = Integer.parseInt(input[2]);
  bf_threads = new BruteForceThread[THREADS];
  MAX_WORD_LENGTH = Integer.parseInt(input[1]);

  for (int i = 0; i < bf_threads.length; i++) {
    bf_threads[i] = new BruteForceThread("bf" + i);
  }

  waitForBFThreads();

  while (!found) {
    if (current_word.length > MAX_WORD_LENGTH)
      break;
    pushWord(current_word.length - 1);
    char[] tmp = new char[current_word.length + 1];
    tmp[tmp.length - 1] = ' ';
    System.arraycopy(current_word, 0, tmp, 0, current_word.↩
        length);
    current_word = tmp;
  }
}
```

bruteForceAttack starts a number of BruteForceThread threads. The function continues after all BruteForceThreads are initialized and ready to start their task. It then enters a while-loop, which executes a pushWord function for each iteration.

The task of pushWord is to simply create and push all possible words of a given length onto a stack of words. The length of the words to push are given by its integer argument. The whole attack is based on letting the main thread create and push words onto a stack, while the BruteForceThreads are pulling words from the stack. When a word is pulled from the stack, it is subject to *PBKDF2*, where the result is used to decrypt the ciphertext. If decryption results in a given plaintext, the password is found.

The dictionaryAttack function is shown in Listing A.2.

Listing A.2: dictionaryAttack function

```java
public void dictionaryAttack(String[] input) {
  File dict = new File(input[1]);
  THREADS = Integer.parseInt(input[2]);

  if (dict.exists()) {
    try {
      fr = new FileReader(dict);
      buf = new BufferedReader(fr);
      start = System.currentTimeMillis();
      for (int i = 0; i < THREADS; i++) {
        new DictionaryThread("dict" + i);
      }
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    }

  } else {
    System.out.println("ERROR: Dictionary file does not exist!");
    printHelp();
    System.exit(0);
  }
}
```

dictionaryAttack reads an input dictionary file into a BufferedReader. It then starts a given number of DictionaryThreads. The DictionaryThreads will read from the BufferedReader in a synchronized way. When a word is read from a DictionaryThread it will be subject to *PBKDF2*, where the result is used to decrypt the ciphertext. If decryption results in a given plaintext, the password is found.

## A.2 Cluster Dictionary Attack

The CDA is written in Java, and is built around the same procedure as the dictionary attack in BFDA. However, the difference lays in the cooperation of multiple computers. To enable a cluster of computers to cooperate, we used the following environment.

### A.2.1 Environment

The environment is based upon a software framework from Apache called *Hadoop*. The main functionality of Hadoop is described below.

#### Apache Hadoop

Apache Hadoop makes it possible for multiple machines to cooperate and run computational work together. Hadoop also provides a distributed file system HDFS, that can store data across multiple cooperating machines [44]. The computational work in Hadoop is organized and distributed using *MapReduce*.

**MapReduce** MapReduce is a programming paradigm introduced by Google. It is designed to process and generate large sets of data using a cluster of machines [45].

In MapReduce, a large set of input data is divided into multiple key/value pairs. The key/value pairs are further distributed to multiple MapReduce tasks running on multiple machines.

A MapReduce task is divided into a *mapper* and a *reducer*. The task of a mapper is to perform an operation on a key/value pair and return a key/value pair as a result to the reducer. The reducer collects key/value pairs from multiple mappers and combine the results into one or more output files.

### A.2.2 Implementation Details

The CDA attack is implemented as a MapReduce problem, with a large dictionary file as the data input set. The dictionary is split into separate key/value pairs, where each value is a single line in the dictionary and the key corresponds to the line's offset within the dictionary.

The key/value pairs are handled by a map function implemented in `CDAMapper`. The map function has the responsibility of checking every word on a single dictionary line. Each line in the dictionary should contain a certain amount of words. This is to enable the map function to run multiple threads at the same time, where each thread checks one or more words. With multiple threads, the attack is able to utilize more CPU power for each running machine in the cluster. A detailed view of the map function, is given in Listing A.3.

Listing A.3: Mapper function in CDAMapper

```java
@Override
public void map(LongWritable key, Text value,
    OutputCollector<Text, LongWritable> output, Reporter ↩
        reporter)
    throws IOException {

  String[] line = value.toString().split(" ");
  String[] line_chunk = new String[WORDS_PER_THREAD];

  // Create threads to check words in line
  for (int i = 0, c = 0; i < (THREADS * WORDS_PER_THREAD)
      && i < line.length; i += WORDS_PER_THREAD, c++) {
    if (line.length - i >= WORDS_PER_THREAD) {
      System.arraycopy(line, i, line_chunk, 0, ↩
          WORDS_PER_THREAD);
    } else {
      System.arraycopy(line, i, line_chunk, 0, line.length - i↩
          );
    }
    dictionary_threads[c] = new DictionaryThread("dict" + i, ↩
        line_chunk);
  }

  // Wait for all threads to finish
  for (DictionaryThread dt : dictionary_threads) {
    try {
```

```
23       dt.thread.join();
24     } catch (InterruptedException e) {
25       e.printStackTrace();
26     }
27   }
28   if (!password.equals("")) {
29     output.collect(new Text("Password is [ " + password
30         + " ]. Found at"),
31         new LongWritable(System.currentTimeMillis()));
32   }
33 }
```

When receiving a key/value pair, the map function first splits the input line into an array of words called line. It then creates a given number of `DictionaryThread` threads and serves each thread a sub array of words from the line array. Each `DictionaryThread` will check all of its incoming words similar to the `DictionaryThread` in BFDA. If the correct password is found, the password will be written to the `sysout` folder on the machine where the mapper is running.

A class named `Processor` initializes the CDA attack by configuring the mapper and reducer tasks. The number of mappers is set equal to the number of nodes used in the cluster. This is to ensure that each machine only runs one mapper at a time. The number of reducers are set to zero, because their behaviour in MapReduce is not needed.

**Notice** The for-loop at Line 10 in Listing A.3 requires the number of words per line, in the dictionary, to be equal to a multiple of the number of threads in use. It is recommended that the input dictionary follows this requirement.

In this occasion, we have created a Bash script, named `dictmaker.sh`, that changes a regular dictionary into an $N$ words-per-line dictionary. The script can be found on the attached disc.

# B

# Attachments

This thesis comes with two available attachments – one digitally uploaded to the DAIM system[1], and one physical DVD.

## B.1 Electronic Attachment

The electronic attachment, uploaded to DAIM, consists of the following files and directories:

**Application** All files and source code of the proof of concept system, i.e. the background library, server and client. A patch file containing security fixes, is also included.

**Report** All files that is used to create this report, including images and tables.

**Other** Scripts and raw data from the experiments.

## B.2 Attached DVD

All the source code generated while working on this thesis, including the source code for the proof of concept client code, can be found on an attached DVD. In addition, all files related to the writing of this document, including images and most of the references, are also included on the same DVD.

[1] `http://daim.idi.ntnu.no/`