Abstract

Preface

The work behind this project report was carried out during the spring semester in 2011 at the Norwegian University of Science and Technology (NTNU), Department of Telematics (ITEM).

Eirik Haver, Eivind Melvold and Pål Ruud

Contents

List of Figures

List of Tables

Listings

Acronyms

ACL Access Control List

AES Advanced Encryption Standard

CA Certification authority

CBC Cipher Block Chaining

DSA Digital Signature Algorithm

DSS Digital Signature Scheme

ECB Electronic Codeboo

FAQ Frequently Asked Questions

IaaS Infrastructure as a Service

IV Initialization Vector

LAFS Least Authority File System

MITM Man-in-the-middle

NIST National Institute of Standards and Technology

PBKDF2 Password-Based Key Derivation Function version 2

PaaS Platform as a Service

PGP Pretty Good Privacy

PKI Public Key Infrastructure

RSA Rivest, Shamir and Adleman

SaaS Software as a Service

SHA Secure Hash Algorithm

SSL Secure Socket Layer

TLS Transport Layer Security

VM Virtual Machine

1

Introduction

1.1 Method

1.2 Outline

The work is presented as per the following chapters:

Chapter 2 provides background knowledge of the technologies and software used.

2

Background

2.1 Security Services

This section explains certain 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 [3, p. 12].

Confidentiality is the art of keeping a message secret from unauthorized parties [3, p. 18]. This can typically be done by either preventing other parties access to the message at all, or making the contents unreadable, for instance by the use of encryption.

Integrity in a security perspective deals with detecting, preventing or recovering a message being changed by an unauthorized party [3].

Authentication is the act for a user, service or similar to prove that he is what he claims to be [3].

Nonrepudiation prevents both sender and receiver of a message from denying a transmitted message, in other words one party can prove the other parties involvement [3].

2.2 Cryptographic Primitives

This section explains the low level security primitives used in this thesis.

2.2.1 Encryption

Encryption is the process of transforming some information into an unreadable form. Encryption is primarily used to enforce Confidentially, but can also be used for other purposes such as authentication. In a very 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, the *ciphertext*. If a good cipher is used, knowledge of the cipher, plaintext and ciphertext should not be enough to obtain the key.

Block-cipher and Stream-cipher are classifications on how a cipher treats data[3, 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 plaintext.

Symmetric encryption is an encryption scheme where the same key is used for both encryption and decryption[3, p. 32]. Advanced Encryption Standard (AES) is a block cipher and is the current standard for symmetric encryption. AES works on a block of 128-bit and support keys of 128, 192 and 256-bit.

The mode of operation used for a symmetric 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 pure AES, which is called the Electronic Codeboo (ECB) mode of operation. The problem with this is that some information of the ciphertext will leak, i.e. the same plaintext will always be the same ciphertext. A more usefull way is *Cipher Block Chaining (CBC)*. In CBC you will need an Initialization Vector (IV) which should be non-predictable, and not reused. The IV is XORed with the first block of plaintext, which again is encrypted with AES. The resulting ciphertext is used as an IV for the next block.

Asymmetric encryption is an encryption scheme where a different key is used for encryption than decryption[3, p. 259]. An asymmetric 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 for the owner of the private key. The downside of asymmetric encryption compared to symmetric is that it requires a larger key and has a larger computational overhead to obtain the same level of confidentiality. The probably best known asymmetric cipher is Rivest, Shamir and Adleman (RSA).

2.2.2 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 [3, p. 335]. It should also be infeasible to find two blocks of data which produce the same hash value (a *collision*).

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

2.3 Applications of cryptographic primitives

2.3.1 Digital Signatures

A digital signature is the digital equivalent of a normal signature, it verifies that an entity approves with or has written a message, the date the signature was made and it should be verifiable by a third party [3, p. 379]. It should logically not be possible or at least unfeasible to fake a digital signature. It is possible to create digital signatures with RSA there is also a standard for digital signatures called Digital Signature Scheme (DSS) which uses Digital Signature Algorithm (DSA) as the actual algorithm.

2.3.2 Digital Certificates and PKI

A digital certificate is the pairing of a digital signature and a public key[3]. By this scheme the services confidentiality, authentication and nonrepudiation can be achieved. Basically a person or other entity has a certificate with some clues about the identity in it, e.g. the 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 practise 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 trusts[3]. Instead trust is made between users by somehow verifying their public key, for instance by meeting face to face. A user can then sign another users key, set a trust level for the user and publish this information to a keyserver. Another user can then calculate a trust to an unknown person based on the trust set by peoples that he trusts.

2.3.3 SSL/TLS

Transport Layer Security (TLS) and its predecessor Secure Socket Layer (SSL) are techniques for obtaining confidentiality, integrity for transfer of files over a network[3]. It does so by a combination of different algorithms and primitives, but a digital certificate is required for authentication.

2.3.4 PBKDF2

Password-Based Key Derivation Function version 2 (PBKDF2) is a key derivation function to create an encryption key based on a password. The point of this is that a password is often something that should be memorable to a person, but what is memorable to a person might be a to short phrase to withstand a brute force attack. What PBKDF2 does is make the process of deriving the key from the password an expensive process in terms of computational power, to make it more resistant to brute force attacks.

2.4 Security Attacks

This section briefly list security attacks relevant to this thesis, as defined by Stallings.

Active and passive attacks are classifications of security attacks, 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 is the art of capturing communication sent between two parties. This information might contain secrets or might for instance leak enough information about an encryption key to make it breakable.

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

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

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 is an active attack where the attacker seeks to make resources unavailable for legit users, i.e. by overloading an application by sending it lots of traffic.

Man-in-the-middle is an attack where an attacker intercepts messages between the communicating parties and then either relay or substitute the intercepted message.

2.4.1 Attacks on cryptographic primitives

Even though cryptographic primitives are designed to be secure, they might have both flaws and be used in an incorrect fashion.

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

Brute-force attack is an attack where you try 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.

2.5 Cloud Computing

In a draft[2] 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.

2.5.1 Service Models

NIST also defines three service models which deals with what kind of service the consumer is able to rent from a provider.

Software as a Service (SaaS) is the capability for a consumer to run the provider's application running on cloud infrastructure, using a thin-client, browser or similar. Gmail[1] can be seen as an example of this.

Platform as a Service (PaaS) is the capability for a consumer to deploy software onto the cloud, but without actually controlling the underlying platform, operating system etc.

Infrastructure as a Service (IaaS) is 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. An example is hiring a Virtual Machine (VM).

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

2.5.2 Deployment Models

The NIST draft also 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 is a cloud infrastructure operated solely for an organization. Which party manages the cloud and where it is located is not given.

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

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

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

2.5.3 Security considerations

There are some considerations when using cloud services from an external provider as opposed to self controlled hardware, software and plattforms. Most notably is that you loose the control of selecting the people which will have physical and digital access to the infrastructure. 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 able possible for a VM to leak information to other VMs on the same host.

2.6 Existing Solutions

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

2.6.1 Dropbox

Dropbox[2] is a commercial application for storing data in the cloud, more specific using Amazons S3 storage service. It claims that files are stored encrypted with AES-256 which can only be decrypted with the users username and password, and that Dropbox employees are not able to access the files of the user[3]. However Dropbox also has a Forgot password feature which means that Dropbox can read the users files if they really want to. Their Frequently Asked Questions (FAQ) does

[2] `http://www.dropbox.com`
[3] `http://www.dropbox.com/help/27`

however say that some people have been successfull in putting truecrypt containers in Dropbox which effectivly makes dropbox secure[4].

2.6.2 Tahoe-LAFS

Tahoe-Least Authority File System (LAFS)[5] is an open source cloud storage file system which does fulfill the requirements set by our problem description in regards to security. In Tahoe-LAFS files are exclusivly encrypted client-side, before beeing uploaded into the cloud. Tahoe-LAFS also uses erasure-coding to obtain redundancy across multiple storage servers.

[4] `http://www.dropbox.com/help/179`
[5] http://www.tahoe-lafs.org

3

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.

3.1 Introduction

The architecture has to support various user functionality. Figure 3.1 exhibits Alice uploading files to the cloud, and thereafter transferring the necessary information to Bob so that he also can gain access to the files.

In the following sections, we will describe how the file storage is organized, and take a closer look at how the different functionality are solved.

3.2 File Storage

The solution proposed in this thesis, is that only a simple key-value store is needed on the server side. This may be extended with an Access Control List (ACL) layer to support user access and other features.

Two types of files exist: immutable and mutable files. The mutable files are used as directories, in the sense that they contain the information needed to access other directories or files in the form of capabilities. A capability is a short alphanumeric string containing all information needed to find, get, read and write a file or folder. This includes an identifier and cryptographic keys.

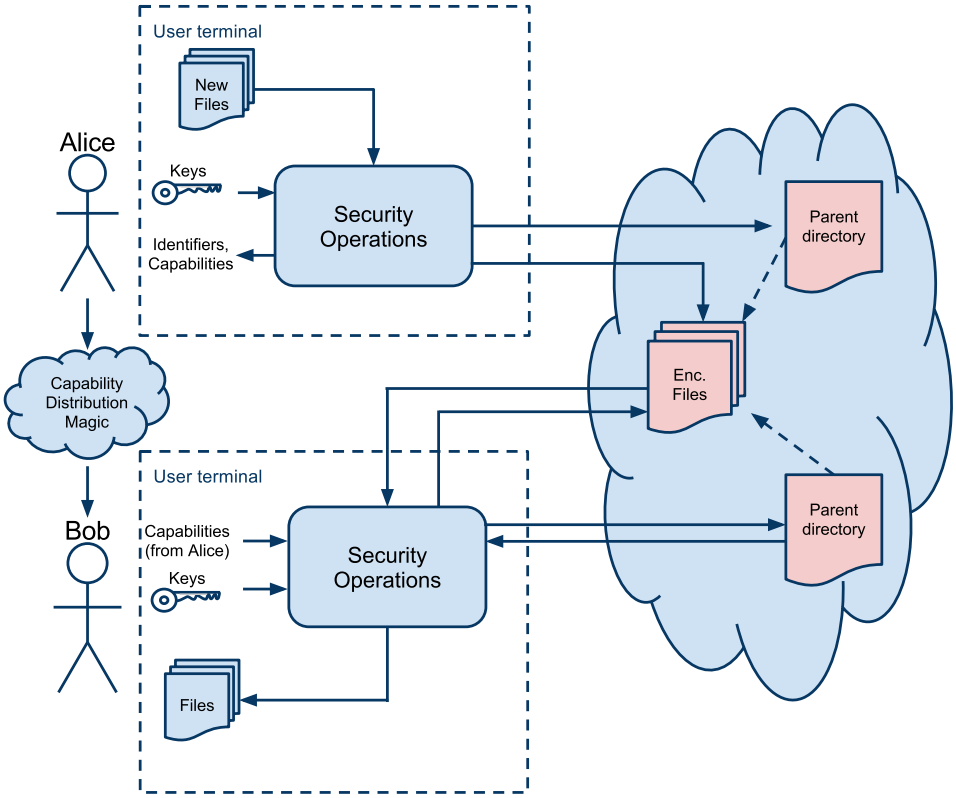

Figure 3.1: Overview of user functionality

In principle, the only operations the key-value store need to support are **PUT**, **GET** and **UPDATE**. The latter one is required to make changes to the "directory" files.

3.2.1 Directory Structure

The file and directory structure can be seen as a directed graph, as illustrated in Figure 3.2, and users can link in folders and files as wanted. This provides a flexible and space-conservative structure that is easily extendible.

Each client[1] keeps a local copy of the identity and write key to the users root folder in a password protected keychain. In the event of loss of one of the clients, a user can change encryption keys (and identifier) for the root folder, and hence effectively block out the compromised clients.

A directory contains two types of data. It contains meta data about it self and in addition zero or more entries. Table 3.1 exhibits the structure of the folder

[1] I.e. the different user terminals.

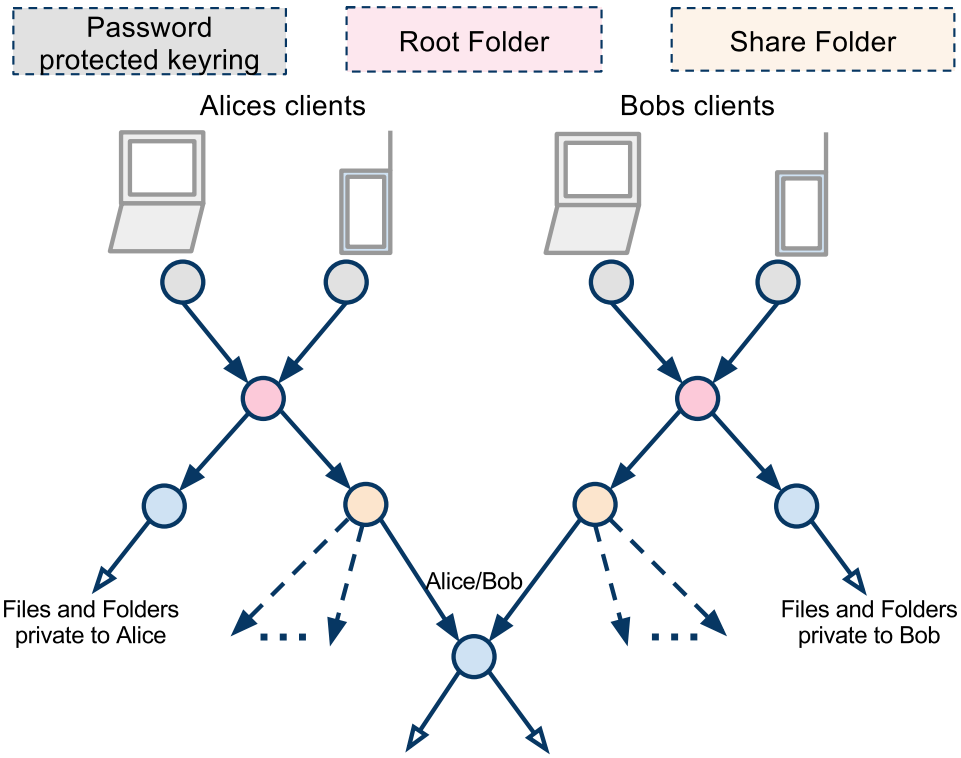

Figure 3.2: File System Structure

entries. The *Alias* field is the name of a specific child. The *storage index* is the look-up value of the corresponding child.

The meta data are described in Table 3.2. The *Storage Index* is a unique value derived from a directory's read key and serves as an identifier for the directory in question. The other elements are used for security purposes, and are described further in Chapter 4.

3.3 ACL/Authentication Layer

There should be some layer present on the application which enforces ACLs and authenticates users which should have access to the stored files. From a security standpoint this is not necessarily required. An attacker would have to be able to guess the storage index(key) and the corresponding encryption key for that index to be able to get any information out of the system. But there are however several reasons why we believe this layer should be implemented.

Table 3.1: The structure of a folder entry

Data	**Comment**
Alias	Human readable name of file/folder
Storage Index	Key to retrive file/folder from server
Write Key	Only for folders, needed to write a file
Read Key	
Verify Key	Only for folders, needed to verify a file

Table 3.2: Meta data for a directory

Public Key	Storage Index	IV	Write Enabler	Signing Key	Signature

3.3.1 Block access

If for whatever reason a storage index and corresponding decryption key is leaked to some third-party, but not to the cloud provider, the layer can prevent an unauthenticated user from retrieving the file. The possibility of also denying authenticated users from retrieving files is also a wanted feature for instance in the event of a stolen/lost terminal where the credentials are saved.

3.3.2 File Deletion

The way we handle folders allows a distinction between write access and read access. The write key can be used to deduce a secret that can be safely leaked to the server which in turn would be used decide if a user has permission to delete a folder. It could also be used for deciding if a user has permission to write to files, but this can be verified by the server in terms of the signature. For immutable files there is no concept of write-access, only read. It is both illogical and impractical to assume to read access should also yield delete-access. A layer which identifies which users first creates a file can by the same method decide who should be able to delete a file. This might also be in the interest of accounting and billing.

3.3.3 Accounting

If this application were to by run by the cloud storage provider, it is important to be able to decide which person should be billed for the storage of files as well as the traffic generated by users uploading and downloading files. For an immutable file the storage can be decided by who created the file, the traffic is billed on whoever retrieved or uploaded a 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 he has done. 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 at least delete them.

3.4 User scenarios

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

3.4.1 Download file

When a user wishes to download a file or directory, all that is needed is the password to unlock the local keychain on the user terminal, as depicted in Figure 3.3. The client sends a GET request with the identifier in question, and the server responds with the encrypted folder. This contains the capabilities needed to locate and decrypt the underlying files and folders. After decrypting the contents, the client again queries the server with the identifier of the wanted file, and there after decrypts it, before displaying it to the user.

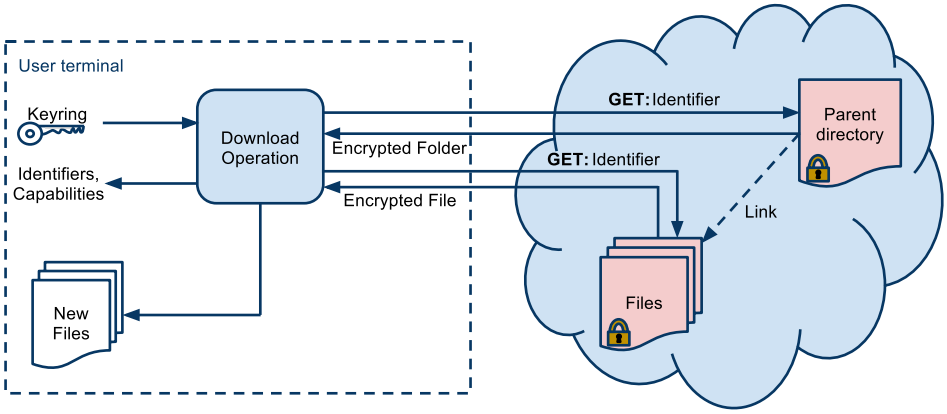

Figure 3.3: Scenario: Downloading of files

3.4.2 Upload Files

Figure 3.4 shows the process of uploading new files. The only information the server in the cloud receives, are identifiers and encrypted containers. The user is also given the opportunity to transfer the corresponding identifiers and capabilities to other users.

Before uploading, the client has to find, download and decrypt the directory the files are to be placed in. This process was described in the previous section. The cryptographic details of the Upload Operation can be found in Section **??**.

3.4.3 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. After the capabilities to this directory has been shared

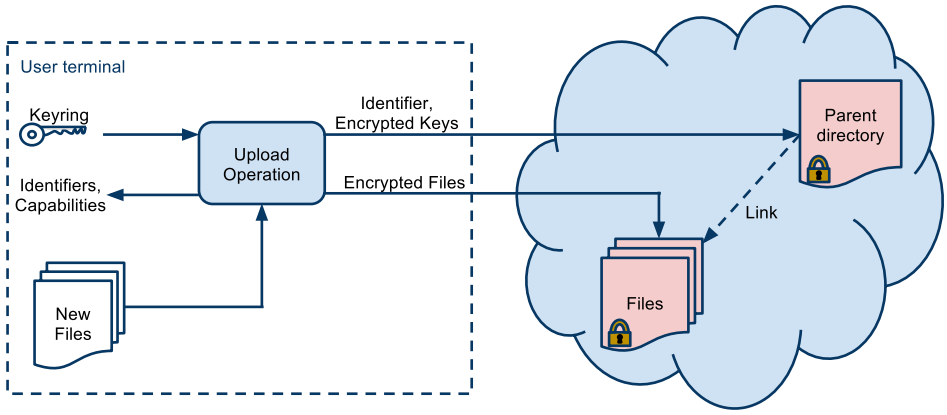

Figure 3.4: Scenario: Uploading of files

with Bob, this directory is going to be the secure channel where they can share folders and files.

Before transferring the capabilities to Bob, Alice links the shared directory to a parent directory, so she can easily find it again at a later time. She can also link files to the shared directory.

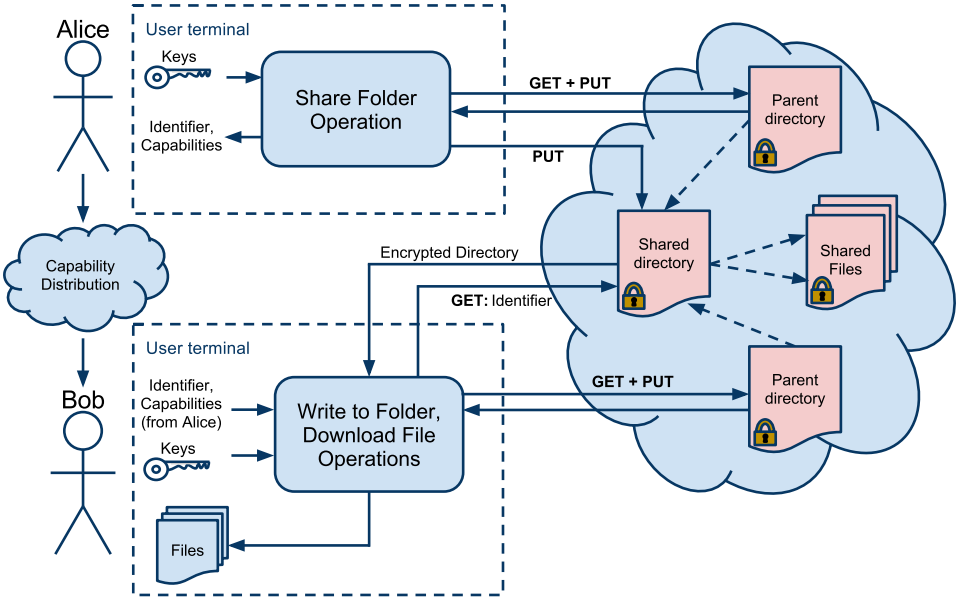

Figure 3.5: Scenario: Sharing files

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 will be described in Chapter TODO.

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 with Bob, she can simply not include the write capability in the parent directory. This will work as intended until Alice wants to write to the folder, thus the write capability has to be stored another place. We will implement this as a special folder under the root folder, and link in all such folders and files the "normal" way here. This process is illustrated in Figure 3.6.

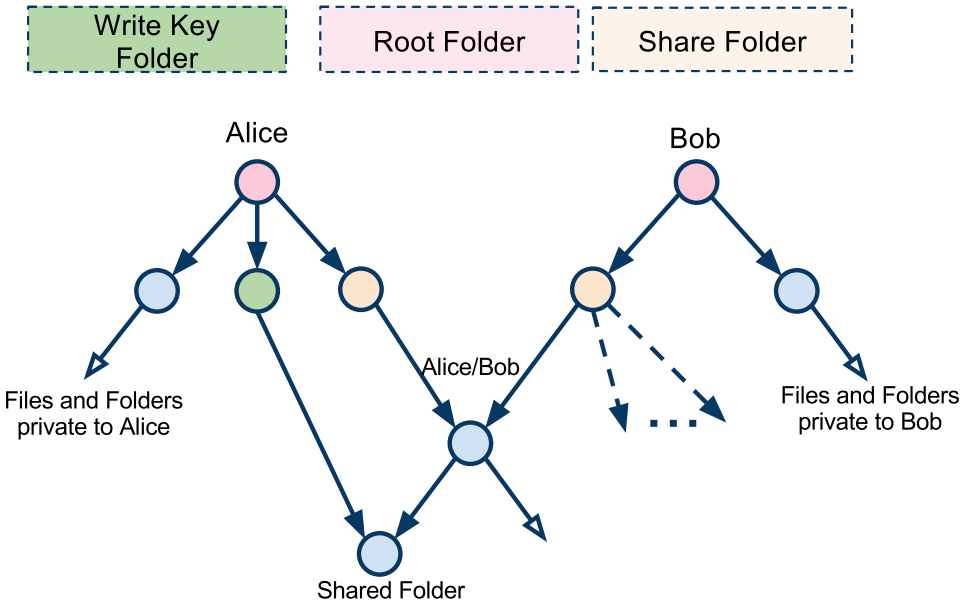

Figure 3.6: Sharing write-protected folders

4

Cryptographic Architecture

This chapter will elaborate on the cryptographic solutions applied to the architectural scheme in Chapter 3. We will take a closer look at how confidentiality, integrity and authentication can be integrated into the proposed architecture.

The first section is a brief introduction explaining the basic security concept used by the application. Following is a detailed description of the cryptographic solutions adapted. The cryptographic solutions are revealed in terms of file and directory operations performed by the application. The chapter ends with a description of a users keychain.

4.1 Security Concept

The basic security concept of the application is to keep a user's remote storage confidential to a third-party storage provider. To solve this, the application encrypt files locally at the users terminal before uploading them to the third-party storage provider. When accessing a file, the application first download the encrypted file and further decrypt it locally. To enable this simple encryption scheme, the user terminal is required to possess at least one cryptographic key. Cryptographic keys are stored in a single keychain at the user terminal.

By initially knowing that files are placed encrypted on a remote server and that the local user possess one or more cryptographic keys, we can continue with a more comprehensive description of the complete cryptographic solution. The details are

explained in terms of the following directory and file operations.

4.2 Directory operations

This section presents the two basic directory operations used by the application. The "create directory" and "open directory" operations are explained as follows.

4.2.1 Create Directory

The create directory operation is divided into two subsequent parts. First, the directory must be created locally at the user's terminal. Second, the directory must further be uploaded to the remote server. Uploading a file or directory is possible only if the user is authenticated.

Creating and uploading a directory from the user terminal is illustrated in Figure 4.1.

Before creating a new directory it is necessary to generate an RSA key-pair. The key-pair is specific to the new directory and consists of a public and private signing key. The signing key is further used to derive a write and a read key for the corresponding directory. The read key is a hashed value of the write key, while the write key equals the hashed value of the signing key.

After key generation, the public key is directly added to the new directory. The directory's storage index is further added as a hash value of the directory's read key. The next value added to the directory is the initialization vector IV. The IV is generated from a counter value stored at the user terminal. It is used to prevent predictable ciphertext, when the directory's entries are encrypted.

The next value added to the directory is the write enabler. The write enabler is an HMAC-SHA-2 value created from the directory's write key and a static chosen text value. It is uploaded as a part of the directory, but can never be downloaded from the server. The write enabler is used by the server to enable users, that possess the directory's write key, to modify the directory. This ensures that only authorized users can modify a directory.

After adding the write enabler, the user must add the directory's private signing key. The signing key should only be available for users with write permission to the directory. It is therefore encrypted with the directory's write key before it is added to the directory. The signing key is encrypted using AES in CBC mode. The next value added is the directory's entries. The entries of a directory should only be visible to users with read permissions to the directory. They are therefore encrypted with the directory's IV and read key prior to insertion. The entries are encrypted using AES in CBC mode. The last value added to the directory is a signed hash value of the encrypted entries. The hash value is encrypted with the signing key using RSA. A signature is needed for other users to know that the directory was created from an authorized user.

After creating a new directory, the directory is further uploaded to the remote server.

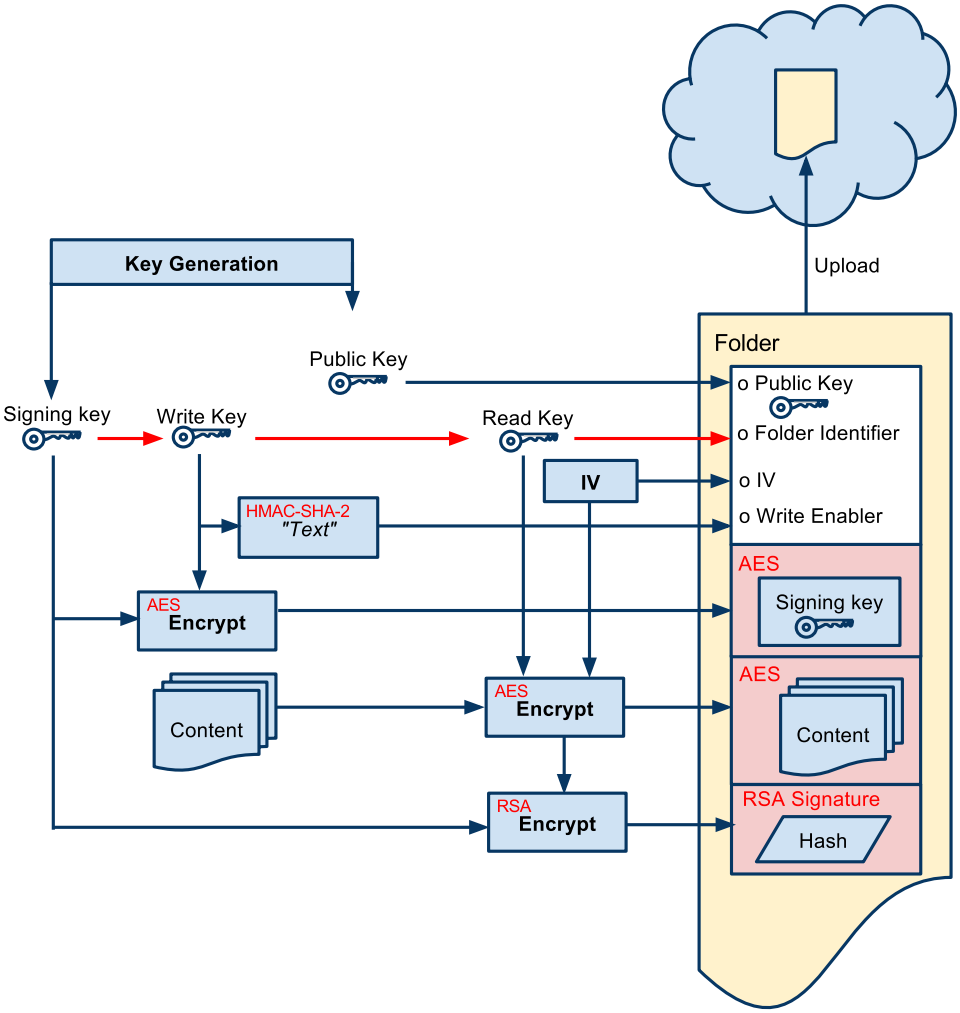

Figure 4.1: Creating and uploading a directory.

It is important to notice that none of the values contained within the directory or any of the cryptographic keys are kept locally by the user. The user can only achieve the public and private keys through the remote directory. The directory's read and write keys can only be achieved through capabilities stored in the remote directory's parent directory. This is true for all directories stored on the server except for the user's root directory. The keys to the root directory are stored in the user's local keychain. The keyring is explained in Section **??**.

4.2.2 Open Directory

This section describes the procedure for opening a directory. Opening a directory involves both downloading, verifying and decrypting the directory. The download and verify procedure is illustrated in Figure 4.2 and described below. A description of the subsequent decryption procedure follows.

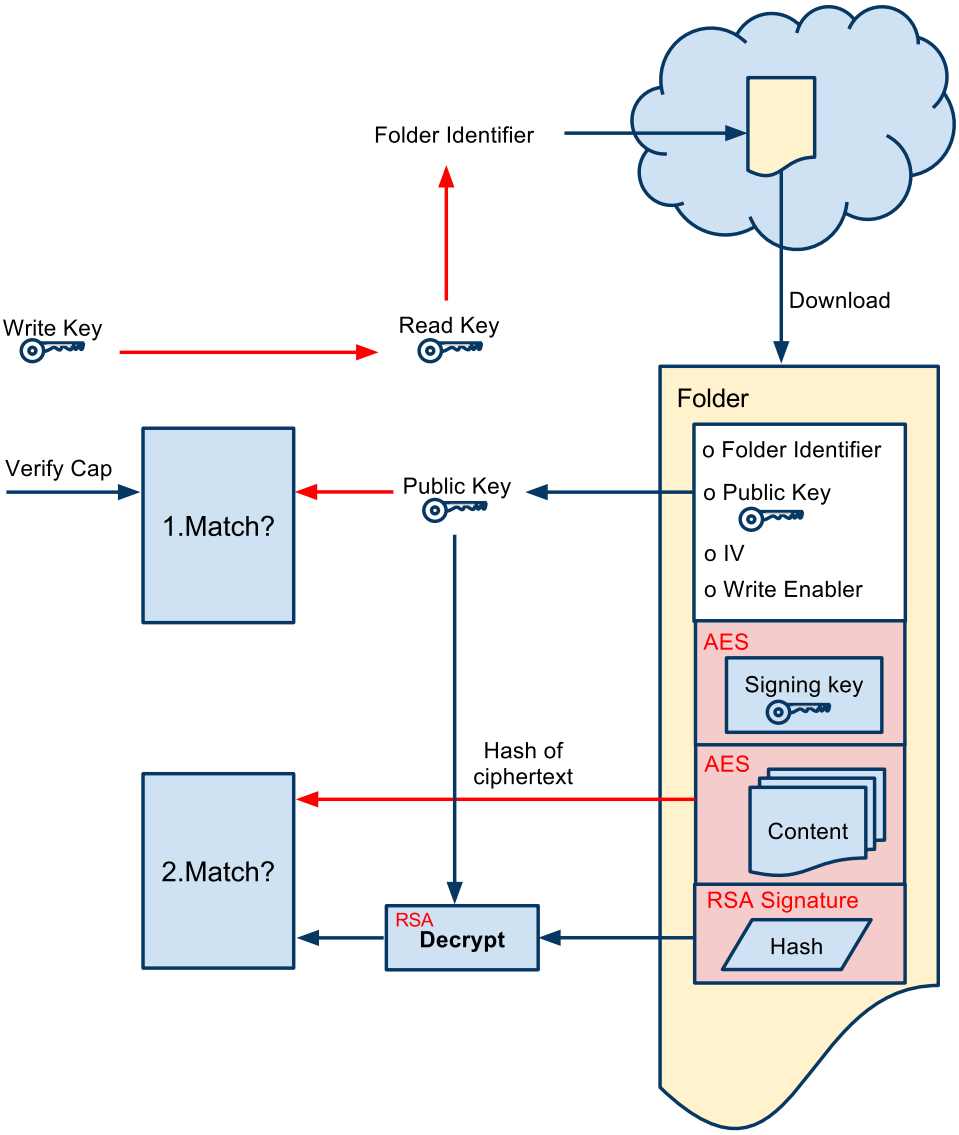

Figure 4.2: Downloading and verifying a remote directory.

First of all, to download a desired directory, the user must be in possession of

a read key specific to that directory. The read key is either created from a hash value of the directory's write key, or obtained from a read capability stored in the parent directory.

The directory's storage index is further obtained. It is either created from the obtained read key, or obtained from the read capability stored in the parent directory. The storage index is a simple hash value of the read key. It is used to localize the specific directory on the remote server. The localized directory is further downloaded to the user's terminal.

When the correct directory is downloaded, it will further exhibit a verification procedure prior to decryption. The verification procedure is illustrated in Figure 4.2.

The purpose of the verification procedure is to check the integrity of the content and to verify that it has been written by an authorized user. If one or neither of these checks are verified, the directory content will not be decrypted.

The first check verifies the correctness of the public key stored within the downloaded directory. The key must be checked to ensure that it belongs to the signing key. If it does, it indicates that an authorized user has signed the directory content. The check is carried out by hashing the directory's public key and comparing it with a verification capability.

If the directory passes the first check, it will further be checked for content integrity. The integrity check ensures that the directory content, in form of entries, has not been tampered with. It is carried out by decrypting a stored encrypted hash value in the directory and comparing it against the hash value of the directory's encrypted entries. The encrypted hash value stored in the directory is decrypted using the previously verified public key. The integrity check is depicted as a part of Figure 4.2.

If the directory passes the verification procedure, the user will decrypt the directory's encrypted entries using the directory specific read key. The decryption procedure is illustrated as a part of Figure 4.3. An IV is additionally used with the read key to carry out the decryption procedure. The IV is easily fetched from the directory prior to decryption.

4.2.3 Modify directory

In addition to read the content of a directory, the user might want to write to it as well. To be able to modify a downloaded directory, the user must be in possession of the directory's write key. The write key is needed to obtain the directory's signing key and write enabler.

The signing key is needed by the user to correctly sign the modified directory. Remember that a directory can only be read by users if it contains a valid signature. Obtaining the signing key from a downloaded directory is shown as a part of Figure 4.3.

The write enabler is generated as a HMAC-SHA-2 value from the write key and a static known text value. It is used to prove the user's write permissions to the receiving ACL layer at the remote server. This ensures that the directory has been modified by an authorized user.

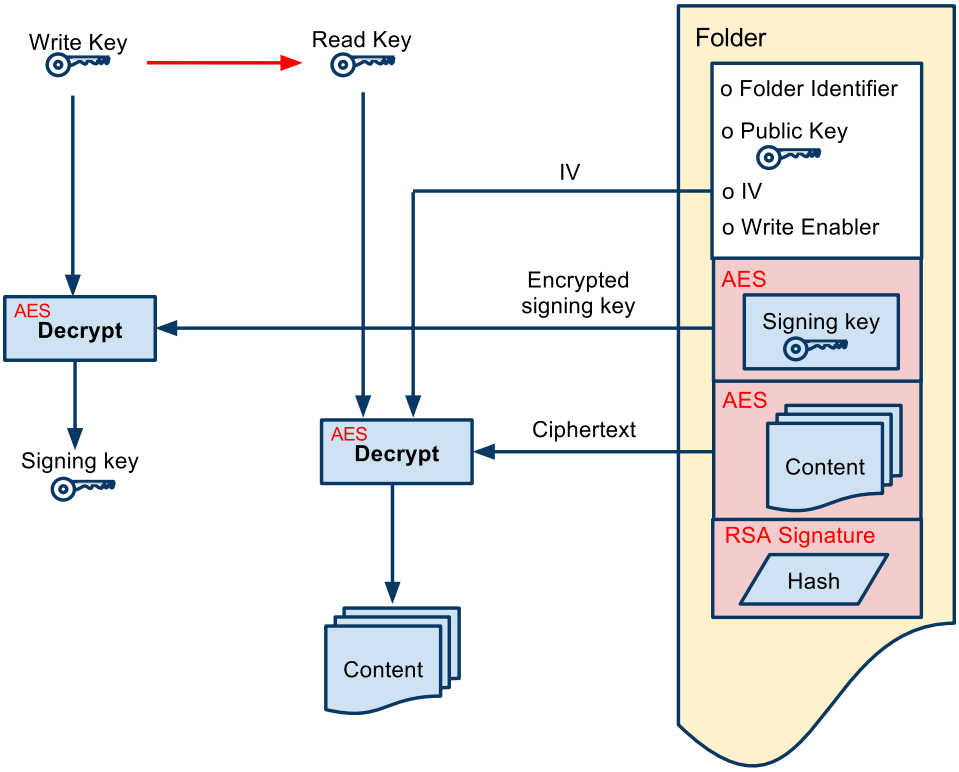

Figure 4.3: Decrypting a remote directory: Decrypting directory content and obtaining the signature key.

The user fetches the encrypted signing key and decrypts it with the write key. This is illustrated in Figure 4.3.

4.3 File Operations

This section describes the elementary file operations supported by the application. The "create file" and "open file" operations are considered.

4.3.1 Create File

In the create file operation, a file must be created locally on the user's terminal prior to upload. The operation is depicted in Figure 4.4 and described as follows.

The user chooses a local file to upload. This file is depicted as the "content" value in Figure 4.4. A cryptographic write key, specific for the new file, is further generated. The key generation algorithm is explained in Section **??**. The write key is further hashed to create a file specific storage index. The storage index is stored in the new file. A file's storage index has the same property as a directory's storage

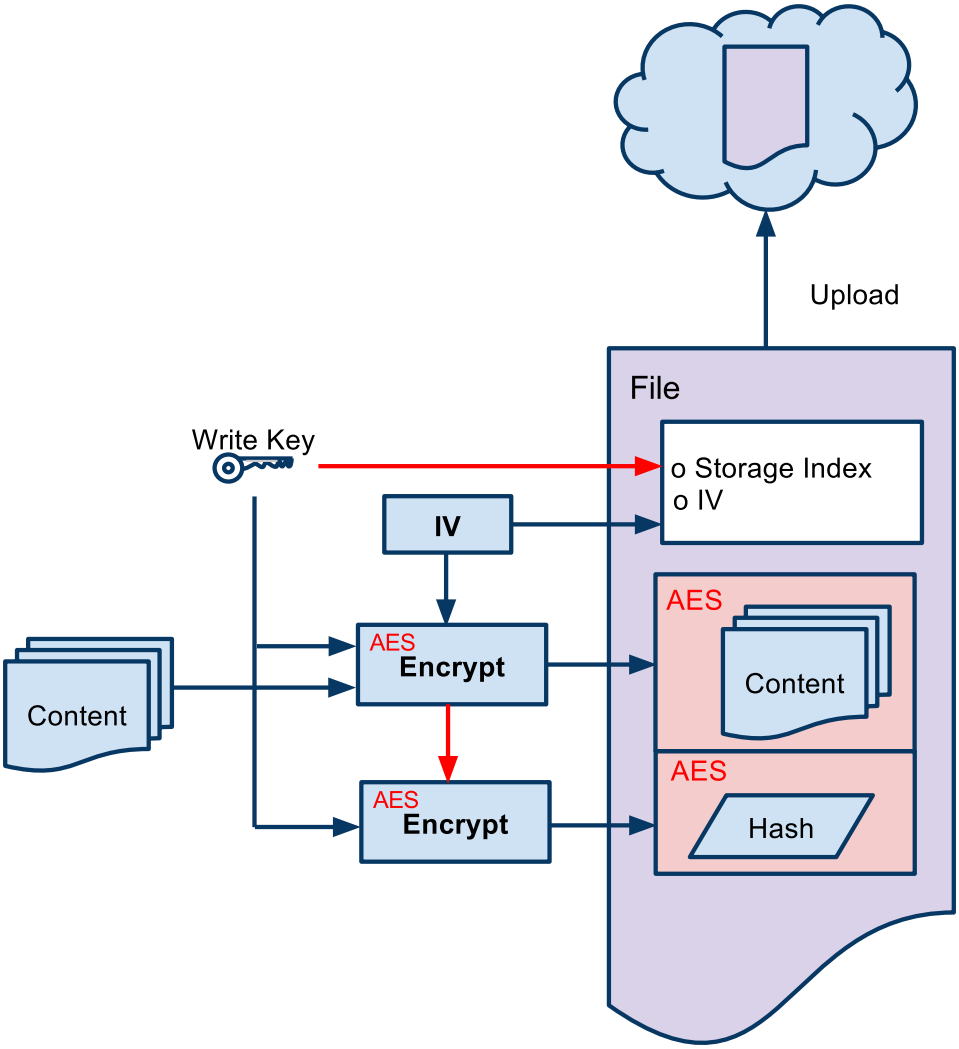

Figure 4.4: Creating and uploading a file.

index. The IV is the next value that is inserted into the new file. It is generated from a counter stored at the user's terminal.

The previously chosen file content is further encrypted with the write key and file IV. The encrypted file content is then inserted to the new file. The last value added to the new file is an encrypted hash value of the encrypted content. The hash value is encrypted using the file's write key. Both the file content and hash value are encrypted using AES in CBC mode.

4.3.2 Open File

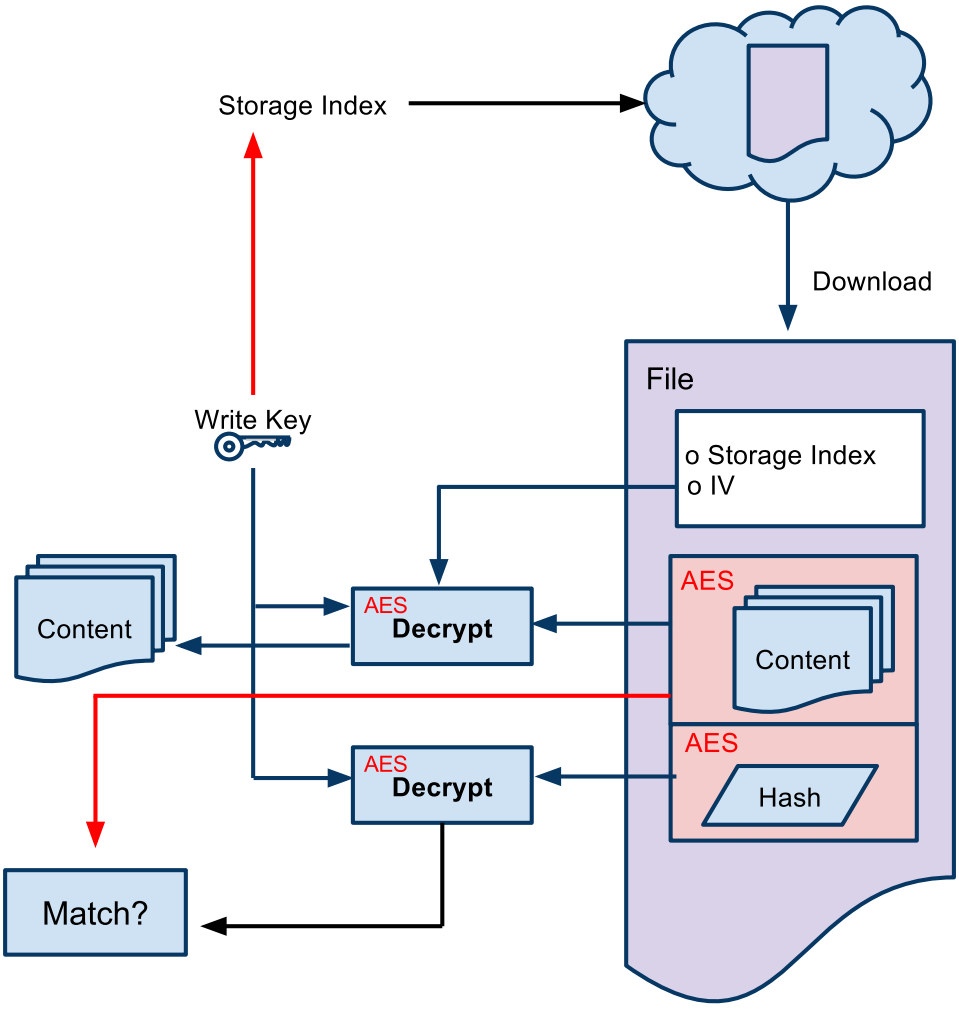

Figure 4.5: Downloading a file from the server

The process of retrieving a file is illustrated in figure 4.5.A file is located by hashing the write key, which in turn is obtained either by knowledge by the user or client or from a parent folder. The ciphertext, the IV and the encrypted hash of the ciphertext is downloaded from the server. The ciphertext is hashed client side and the encrypted hash from the server is decrypted with the write key, if these two outputs match the ciphertext has not been tampered with. The ciphertext is then decrypted using the write key and IV to obtain the file contents.

4.4 Keychain

As stated, each client keeps a local copy of the identity and write key to the personal root folder in a password protected keychain. This is done to be able to encrypt the root folder with a key that is not based on a password, but completely randomly generated. In addition, this means that no special scheme is needed for the root folder, as we can use the same procedures as for "normal" folders as defined in the previous sections.

The strength of a password is related to its length and its randomness properties. Passwords shorter than 10 characters are usually considered to be weak [4]. In the event of loosing one of the clients, and thus the 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 [1].

4.5 Choice of cryptographic primitives

The scheme which has been described in this section requires three primitives; a cryptographic hash function, a symmetric cipher and an asymmetric cipher which can be used for digital signatures. The choices should reflect the following security demands of our application, all with the assumption that the possible attacker has access to vast quantities of computing power.

1. It should be infeasible to break confidentiality for any file or folder
2. It should be infeasible to break integrity for any file or folder

4.5.1 Symmetric Cipher

For a symmetric cipher the choice is pretty simple; AES. AES is a standard, has been around for a long time, and does not have any serious security issues as long as it is used correctly.

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

5

Implementation

This chapter describes the implementation of our proof-of-concept client and server.

5.1 Server

5.1.1 Communication with Client

5.1.2 Upload/Download Functionality

5.1.3 ACL functionality

5.2 Client

5.2.1 Structure

5.2.2 Communication with Server

5.2.3 Cryptography

5.2.4 Sharing

5.2.5 Adding a new client

5.2.6 User Interface

5.2.7 Secure storage of files on Client

6

Results

7

Discussion

7.1 Complexity

7.1.1 Keys

7.1.2 Files

7.1.3 Folders

7.1.4 Sharing

7.2 Security

This thesis is written with the basis that we are hiring storage from an untrusted provider, and the security of the application should primarily reflect that. With this starting point we can assume that all data stored on the server is obtainable by the provider.

7.2.1 Passive attacks

The provider will have access to all data on the server, and also know what data stored is part of files and what is part of folders. 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. In other words a folder is attackable both trough the asymmetric and the symmetric cipher used. If the confidentiality of a folder is

breached this will also lead to the attacker being able to decrypt and subfolders or filer, in contrast the breach of confidentiality of a file will only compromise that specific file. It is also important to note that users will use a root folder which any other file or folder relies on, if the confidentiality of this folder is breached, all files are effectively compromised.

Brute Force attacks

RSA attacks

AES attacks

7.2.2 Active attacks

7.2.3 Terminal security

Given enough users, at some point there will be a users who looses his terminal, for instance a mobile phone, a terminal might also be broken into without the users knowledge.

7.3 Performance

7.4 Omitted features

8

Conclusion and Future Work

Bibliography

[1] 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`.

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

[4] 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`.

Appendices