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 storage as a cloud service to their customers. This is a preferable solution to regular storage in terms of low hardware costs, reliability, scalability and capacity.

However, the idea of storing customer data by an untrusted cloud provider introduces the issue of data privacy and integrity. The customer is no longer in position to control the physical access to the stored data, and is therefore not guaranteed data privacy or integrity by the cloud provider.

To solve this problem, we have proposed a solution that ensures privacy and integrity of customers' data stored by untrusted cloud providers. The proposed solution does also support sharing of private data among customers of the same cloud provider. The solution has further been implemented as a proof of concept.

When implementing the solution, we first created an underlying framework to support the proposed and necessary cryptographic functionality. The framework was further utilized to implement the final application.

Application Results:

Conclusion:

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

Abstract I

Preface III

List of Figures IX

List of Tables XI

Listings XIII

Acronyms XV

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Related Work . . . 2
- 1.3 Scope and Objectives . . . 3
- 1.4 Limitations . . . 3
- 1.5 Methodology . . . 4
- 1.6 Outline . . . 5

2 Background 7
- 2.1 Security Services . . . 7
- 2.2 Cryptographic Primitives . . . 8
  - 2.2.1 Encryption . . . 8
  - 2.2.2 Cryptographic Hash Functions . . . 9
- 2.3 Applications of Cryptographic Primitives . . . 9
  - 2.3.1 Digital Signatures . . . 9
  - 2.3.2 Digital Certificates and PKI . . . 10
  - 2.3.3 SSL/TLS . . . 10
  - 2.3.4 PBKDF2 . . . 10
- 2.4 Security Attacks . . . 11
  - 2.4.1 Attacks on Cryptographic Primitives . . . 11
- 2.5 Cloud Computing . . . 12
  - 2.5.1 Service Models . . . 12
  - 2.5.2 Deployment Models . . . 12

2.5.3 Security Considerations in Cloud Computing . . . 13
2.6 Research . . . 13
2.6.1 Privacy as a Service . . . 13
2.6.2 Privacy Manager . . . 15
2.6.3 Trusted Cloud Computing Platform . . . 15
2.6.4 Cryptographic Cloud Storage . . . 17
2.7 Existing Solutions . . . 18
2.7.1 Dropbox . . . 18
2.7.2 Tahoe-LAFS . . . 19
2.7.3 Wuala . . . 20

3 Technical Procedure **21**
3.1 Architectural Overview . . . 21
3.1.1 File Storage . . . 22
3.1.2 ACL/Authentication Layer . . . 23
3.1.3 User Scenarios . . . 24
3.1.4 Constraints . . . 27
3.2 Cryptographic Architecture . . . 27
3.2.1 Security Concepts . . . 28
3.2.2 File and Directory Operations . . . 29
3.2.3 Recommendations for Cryptographic Primitives . . . 32
3.3 Server Implementation . . . 35
3.3.1 Communication and Architectural Patterns . . . 35
3.3.2 Environment . . . 37
3.3.3 Implementation Details . . . 37
3.4 Client Implementation - Android . . . 39
3.4.1 Environment . . . 39
3.4.2 Architectural Patterns . . . 40
3.4.3 Implementation Details . . . 40
3.4.4 Sharing . . . 43
3.4.5 Adding a New Client . . . 43
3.4.6 Securing the Client . . . 43
3.4.7 User Interface . . . 44

4 Experimental Procedure **49**
4.1 Performance . . . 49
4.1.1 What is Measured . . . 50
4.1.2 How we Measure . . . 50
4.1.3 Eliminating Bottlenecks on Android Devices . . . 51
4.1.4 Sources of Error . . . 51
4.2 Security . . . 51

5 Results **57**
5.1 Client Performance . . . 57
5.1.1 Files . . . 57
5.1.2 Folders . . . 58
5.2 Brute Force Local Keyring . . . 58
5.2.1 Brute Force and Dictionary Attack (BFDA) . . . 60
5.2.2 Cluster Dictionary Attack (CDA) . . . 61

6 Discussion **63**
6.1 Cryptographic Scheme . . . 63
6.1.1 Influence of Tahoe-LAFS . . . 63
6.1.2 Sharing . . . 64
6.1.3 Deletion of Files . . . 64
6.1.4 Verification of Files . . . 65
6.1.5 Version Control System . . . 66
6.1.6 Deduplication . . . 67
6.2 Implementation . . . 68
6.2.1 Choice of Use Case . . . 68
6.2.2 Choice of Key Distribution . . . 68
6.2.3 Choice of Cryptographic Primitives . . . 69
6.3 Performance . . . 69
6.3.1 Accuracy of Measurements . . . 70
6.4 Security . . . 70
6.4.1 Security of the Cryptographic Scheme . . . 70
6.4.2 Client Security . . . 71
6.5 Additional Features . . . 73

7 Conclusion and Future Work **75**
7.1 Compared with Other Solutions . . . 75

Appendices **83**

A Other Relevant Implementations **83**
A.1 Brute Force and Dictionary Attack . . . 83
A.1.1 Implementation Details . . . 83
A.2 Cluster Dictionary Attack . . . 85
A.2.1 Environment . . . 85
A.2.2 Implementation Details . . . 86

B Attachments **89**
B.1 Electronic Attachment . . . 89
B.2 Attached DVD . . . 89

List of Figures

2.1 System model of PasS 14
2.2 System architecture of TCCP 16
2.3 Cryptographic cloud storage, customer scenario. 17
2.4 Cryptographic cloud storage, enterprise scenario. 18
2.5 Tahoe-LAFS: Insertion of a new file 19

3.1 Overview of user functionality 22
3.2 File system structure 23
3.3 Scenario: Downloading a file 25
3.4 Scenario: Uploading a file 25
3.5 Scenario: Sharing files 26
3.6 Sharing write-protected folders 27
3.7 Behind the scenes: Uploading a file 30
3.8 Behind the scenes: Downloading a file 31
3.9 Behind the scenes: Creating a directory 32
3.10 Verifying a directory 33
3.11 Decrypting the contents of a directory and obtaining the signing key 34
3.12 Architectural layers in the server application. 36
3.13 Server module structure 38
3.14 Cryptographic entities and their relations 41
3.15 Serialized form of a capability 41
3.16 Establishing a share by copying the key 44
3.17 Establishing a share by using barcodes 45
3.18 Main screen of the client application 46
3.19 Browsing the cloud storage from the client 46
3.20 Context menu showing actions available for items stored 47

4.1 The keyring format with encrypted fields shaded in blue 52

5.1 Results from running brute force and dictionary attacks against a local encrypted keyring. 60

6.1 Theoretical cycle in the directory graph 65
6.2 Hash tree of a file 66
6.3 Tahoe-LAFS deduplication scheme 68

List of Tables

3.1 The contents of a Capability 28
3.2 The REST interface of the server application. 36

4.1 HTC Desire Specifications 49
4.2 HTC Hero Specifications 50
4.3 Test Computer Specifications 50
4.4 Hardware Specifications for Computer Executing the Brute Force and Dictionary Attack (BFDA) 53
4.5 Hardware Specifications for Cluster Instances Executing the Cluster Dictionary Attack (CDA) 53

5.1 File upload/download on CSV 58
5.2 File upload/download on CSV with encryption and hashing disabled 58
5.3 Speed of individual operation on HTC Desire with a 4,38 MB file . . 58
5.4 Create a blank folder 59
5.5 Serialize the contents of a folder with n*86 bytes of data 59
5.6 Encrypt and sign the contents of a folder with n*86 bytes of data . . 59
5.7 Verify a folder with n*86 bytes of data 59
5.8 Speed results of running BFDA. Results are given for different numbers of Password-Based Key Derivation Function version 2 (PBKDF2) iterations. Speed results are measured in passwords checked per second. 60

Listings

3.1 URL mapping in fileserver.py 38
3.2 Pipe and filter upload of a file 42
4.1 Running local brute force attack 53
4.2 Running local dictionary attack 54
4.3 Starting Hadoop Cluster with HDFS 54
4.4 Copying files into HDFS 54
4.5 Executing the CDA Attack 54
A.1 bruteForceAttack function 83
A.2 dictionaryAttack function 84
A.3 Mapper function in CDAMapper 86

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
CG	Credential Generator
CPU	Central Processing Unit
CSV	Cloud Storage Vault
CTR	Counter
DP	Data Processor
DRY	Don't Repeat Yourself
DSA	Digital Signature Algorithm
DSS	Digital Signature Scheme
DV	Data Verifier
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 Autentication Code
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
PasS	Privacy as a Service
PEP	Python Enhancement Proposal
PGP	Pretty Good Privacy
PKI	Public Key Infrastructure
PoC	Proof-of-concept
PRNG	Pseudorandom Number Generator
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
TG	Token Generator
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
WSGI	Web Server Gateway Interface

1

Introduction

The term *Cloud Computing*, and the common shorthand *The Cloud*, 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 of businesses 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, picture collections, file storage and a lot more for free, and available at anytime from anywhere [3, 4].

However, this often comes at a cost of reduced privacy, as the control over the hosting environment is lost. Users is 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 at the cloud provider. Should it be sufficient to trust the security policies of the cloud providers, or is it possible to handle the privacy issues locally before giving away the files?

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 a customer's data is kept private and secure from disloyal employees at the storage provider.

To solve this issue, there has lately been introduced numerous applications and architectures [5, 6, 7, 8] providing solutions to ensure privacy and integrity of customer data stored at the *insecure* cloud provider. However, all of these alternative systems are missing one or more features towards being, as we will define it, a completely secure storage solution.

This situation has given motivation to implement a proof of concept application, that fulfills all of the criteria for what we define as a secure cloud storage service.

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 quite 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 if you can trust, 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 2.6, we present four papers [8, 9, 7, 10] 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 2.7, we present three commercially and publicly available software services that relates to this way of thinking.

One of these applications, Tahoe-Least Authority File System (LAFS) [6], 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

There are two main objectives of this thesis. 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. Data should be encrypted in a safe environment trusted by the owner, before it is stored on the server.
2. It should be possible to verify the integrity of the stored data.
3. Only authorized people should have access to the data.
4. The storage scheme should be documented in detail, such that users can easily understand the scheme and accept/reject it on that basis.
5. An implementation of the storage scheme should be Open Source.
6. An intrusion of the server should not affect the security of the stored files.

The cryptographic scheme should, in addition to the mentioned security, provide the possibility of sharing stored data between multiple cloud storage customers. The security requirements above should also be valid for the shared data. The underlying scheme for sharing data should be applicable for both enterprise and regular customer scenarios.

The second main objective is to implement the proposed scheme as part of a proof of concept application for Android devices. The application should be designed for regular customers, although the underlying cryptographic scheme should additionally support enterprise scenarios. It is further important that the server side of the application is compatible with as many existing cloud storage providers as possible.

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 complete 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) in itself will not get special attention
- 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 [11]. There does not exist a SCRUM *product owner* for the system we will create, nor do we fulfill the requirements and characteristics [11] of a traditional SCRUM *team*. 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 [12] 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 fulfill the requirements set for the specific task, we will choose to utilize the library instead of developing similar code by ourselves.

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

1.6 Outline

This thesis is presented as per the following chapters:

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 performs 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. In addition, associated functionality to a secure cloud storage system, that has not been mentioned earlier, is presented and discussed.

Chapter 7 – Conclusion 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 provided.

2

Background

The basis and underlying technologies for a secure storage service in the cloud, are quite numerous and quite often complex.

In the following sections, we will go through the security services, cryptographic services and attacks that are relevant to such a service. In addition, we present related research and existing solutions available at the time of writing.

This chapter forms the basis for the architecture of the proposed solution presented in this thesis.

2.1 Security Services

This section explains the security services and technology 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 [13, p. 12].

Confidentiality Confidentiality is the act of keeping a message secret from unauthorized parties [13, 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 [13].

Availability The property of a system being accessible and usable upon demand by an authorized system entity, are defined by the availability service [13].

Authentication Authentication is the act of a user, service or similar to prove that he is what he claims to be [13].

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 [13].

2.2 Cryptographic Primitives

This section describes the low level security primitives used in the results of this thesis.

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 *Cryptographic Pseudorandom Number Generator (PRNG)* deterministically produce numbers based on a seed, and it should be infeasible to determine the next number without knowing the seed [14, p. 140].

2.2.1 Encryption

Encryption is the process of transforming some information into an unreadable form. It is primarily used to enforce confidentiality, but can also be used for other purposes, e.g. authentication.

In its 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, i.e. the *ciphertext*. If a strong cipher is used, knowledge of the cipher, or multiple plaintext and multiple ciphertext, 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 on how a cipher treats data [13, 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 Encryption Symmetric-key encryption is an encryption scheme where the same key is used for both encryption and decryption [13, p. 32]. The Advanced Encryption Standard (AES) is a block cipher and is the current standard for symmetric encryption. The AES works on a block of 128 bits at a time, and support keys with length of 128, 192 and 256 bits.

The Mode of Operation The mode of operation used for a 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 pure AES, 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.

An other mode is *Cipher Block Chaining (CBC)*. In CBC, a non-predictable and not reused Initialization Vector (IV) is used. 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 [13, p. 183], and so on.

Asymmetric Encryption Asymmetric key encryption is an encryption scheme where different keys are used for encryption and decryption [13, 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 that only the owner of the private key can decrypt.

The downside of asymmetric encryption compared to symmetric is that it requires a larger key, and that it has a larger computational overhead to obtain the same level of confidentiality as comparable symmetric-key encryption. 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 [13, 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, 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 [13, p. 379].

It should logically not be possible, or at least unfeasible, to fake a digital signature.

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.

2.3.2 Digital Certificates and PKI

A digital certificate is the pairing of a digital signature and a public key [13]. By this scheme, the services confidentiality, authentication and non-repudiation can be achieved.

For example, a person 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 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 [13]. 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 key server.

Another user can then calculate a trust on an unknown person, based on the trust set by peoples that he trusts, from information located on publicly available key servers.

2.3.3 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 [13]. 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).

2.3.4 PBKDF2

PBKDF2 is a key derivation function used to create an encryption key based on a password. The key point of this, is that a password is often something that a person has to be able to recall from memory. But, a password phrase that is memorable to a person, might be too short to withstand a dictionary or even a brute force attack.

What PBKDF2 does, is to 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. This feature is known as *key stretching*.

The efficiency of the key derivation, is dependent on the number of process iterations chosen in the implementation of the function.

2.4 Security Attacks

This section briefly list security attacks relevant to this thesis, as defined by Stallings [13, 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 make it breakable.

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 it lots of traffic.

Man-in-the-middle Man-in-the-middle (MITM) 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 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.

2.5 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*.

2.5.1 Service Models

The NIST also defines three service models which deals with what kind of service the consumer can rent from a provider.

Software as a Service (SaaS) The capability for a consumer to run the provider's application running 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.

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).

2.5.2 Deployment Models

The NIST draft [2] 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 A private cloud is a cloud infrastructure operated solely for an organization. Which party manages the cloud, and where it is located is not defined.

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

Community Cloud A 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 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.

2.5.3 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 [15]. 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 [15].

2.6 Research

This section will elaborate on selected previous research concerning privacy within cloud computing. The research 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 this thesis is set out to solve, although from a different angle. A criterion of the scheme presented in this thesis, 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 seeks to secure the cloud server itself, and the last one argues for building a secure system on top of non-trusted cloud servers.

2.6.1 Privacy as a Service

A concept entitled Privacy as a Service (PasS), was suggested in 2009 [8]. 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.

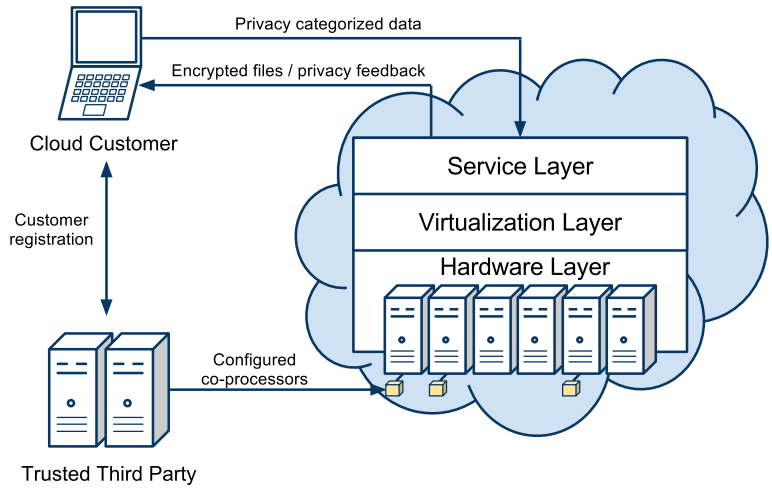

Figure 2.1: System model of PasS

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 a customer's sensitive data.

The cryptographic coprocessors are used in the cloud because they are tamper-proof against physical attacks. The coprocessors are pre-configured by the TTP before they are installed. By using this procedure, the TTP provides a safe computational environment for the cloud customer, 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 TTP, coprocessors and the cloud customer. The customer's key pair 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 none-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.

2.6.2 Privacy Manager

In 2009, HP Labs proposed a way to manage and control a user's private data, stored and processed in a cloud facility [9]. 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. [9] mention the following obfuscation methods:

- Yao's protocol for secure two-party computation [16]
- Gentry's homomorphic encryption scheme [17]
- Narayanan and Schmatikov's obfuscation method [18]

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 [18].

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) is an example of an organization developing and providing trusted computing solutions [19]. A tamper-proof piece of hardware called a Trusted Platform Module (TPM) is recommended [9], 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.

2.6.3 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 [7]. 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 Trusted Platform Module (TPM) chip designed by the Trusted Computing Group. The TPM contains a private/public key pair that it uses to uniquely identify itself. The public key is additionally signed by the manufacturer to guarantee correctness of the TPM chip.

With this in mind, TCCP is based upon a *remote attestation scheme*. The scheme enables a network entity to verify whether another remote entity runs a TPM chip or not. A detailed description of the remote attestation scheme is given by Santos et al. [7].

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.

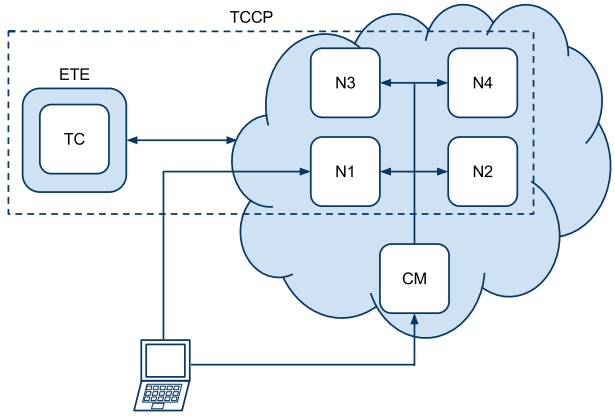

Figure 2.2: System architecture of TCCP

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. Details about the design of the trusted VM monitor, are given by Santos et al. [7].

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. [7].

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.

2.6.4 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 [10]. 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.

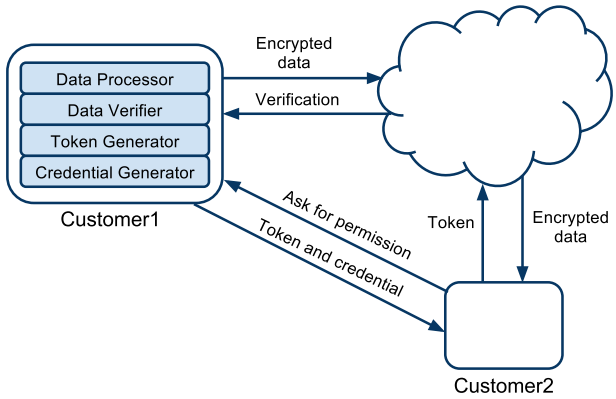

Figure 2.3: Cryptographic cloud storage, customer scenario.

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 (DP)
Processes data before it is sent to the cloud.

Data Verifier (DV)
Checks whether data stored in the cloud has been tampered with.

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

Credential Generator (CG)
Responsible for creating and distributing access policies.

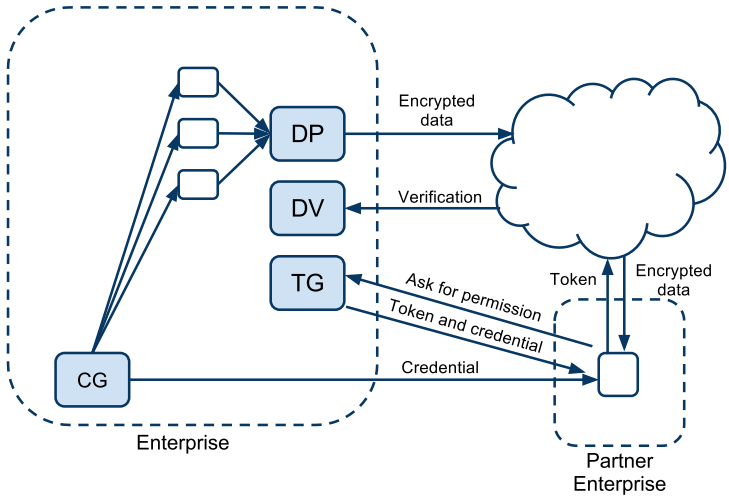

Figure 2.4: Cryptographic cloud storage, enterprise scenario.

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

2.7 Existing Solutions

There are a number of existing 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.

2.7.1 Dropbox

Dropbox[2] 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 strong encryption and strict access control [5], but has received criticism for its lack of security [21]. Among these concerns, is the *Forgotten Password* feature, which implies that Dropbox can read the users files if they really want to – because they have the password – and that the encryption is performed server-side.

In addition, Dropbox is not Open Source, and hence one has no way of verifying that the security features actually work as claimed.

[2] `http://www.dropbox.com/`

2.7.2 Tahoe-LAFS

Tahoe-LAFS[3] is an open source, distributed and secure cloud storage file system, which does fulfill the criteria set 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* [6].

In Tahoe-LAFS, files are exclusively encrypted client-side, then split up using *Erasure-coding*, before being uploaded into the cloud[4], as illustrated in Figure 2.5.

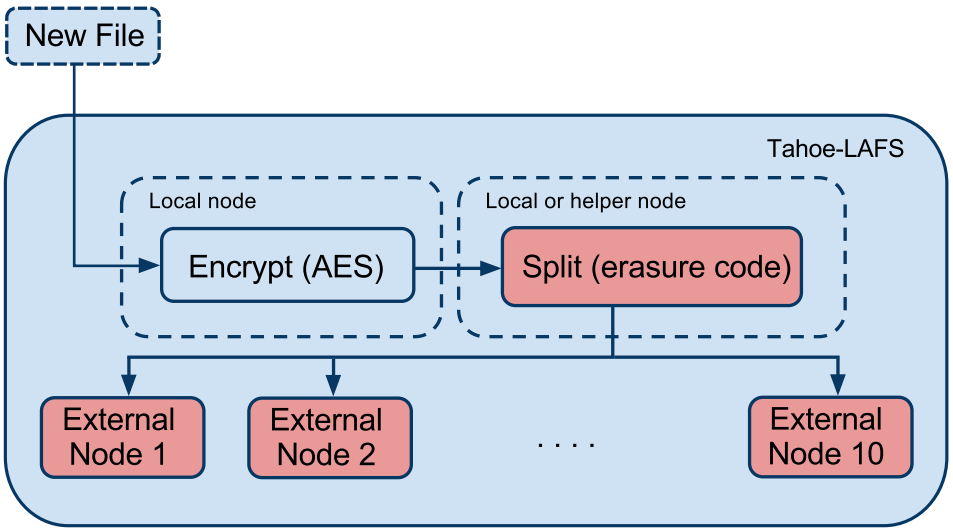

Figure 2.5: Tahoe-LAFS: Insertion of a new file

Architecture of Tahoe-LAFS

Tahoe-LAFS has a three layer architecture: the key-value store, the filesystem, and the application [6].

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 a file, and how to verify it. Nodes in the grid learn about each other through an *introducer*.

The **filesystem** 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 forms of capabilities is available for each file, read-only and read-write, and these can be distributed to e.g. share a file with friends.

[3] `http://www.tahoe-lafs.org/`
[4] The *cloud* in the Tahoe-LAFS sense, often refers to other nodes in a so called *friend net*.

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, i.e. it cannot be modified, and can be read repeatedly. 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 [22]. The original C characters can then be recovered from a subset of the N symbols.

2.7.3 Wuala

Wuala[5] is a closed source secure cloud storage file system, that seemingly operates very similar to Tahoe-LAFS.

The authors have released a paper on a cryptographic tree structure for file systems, called Cryptree [23], and has strong focus on reliability by both providing central servers, in addition to a *P2P cloud* of Wuala users that has donated capacity to the system.

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

3

Technical Procedure

This chapter will describe a proposed 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 a separate server and client functionality.

The chapter will start by giving an overview of the proposed architectural scheme 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 designed by us, 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 to other users to realize sharing of files or directories.

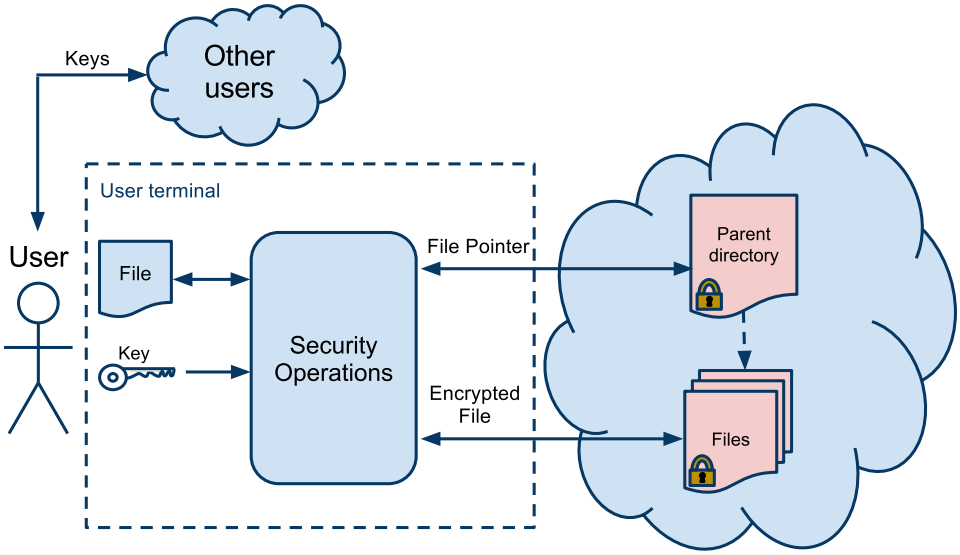

Figure 3.1: Overview of user functionality

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 however, 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 utilized as a normal file.

A user will need certain information to be able to reach and read a file object, 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 possibly write to the file object.

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 capabilities of these file objects. 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 from multiple directory, 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 needs to be stored and protected by the client in a password protected *keyring*. The resulting structure is a directed graph, as illustrated in Figure 3.2.

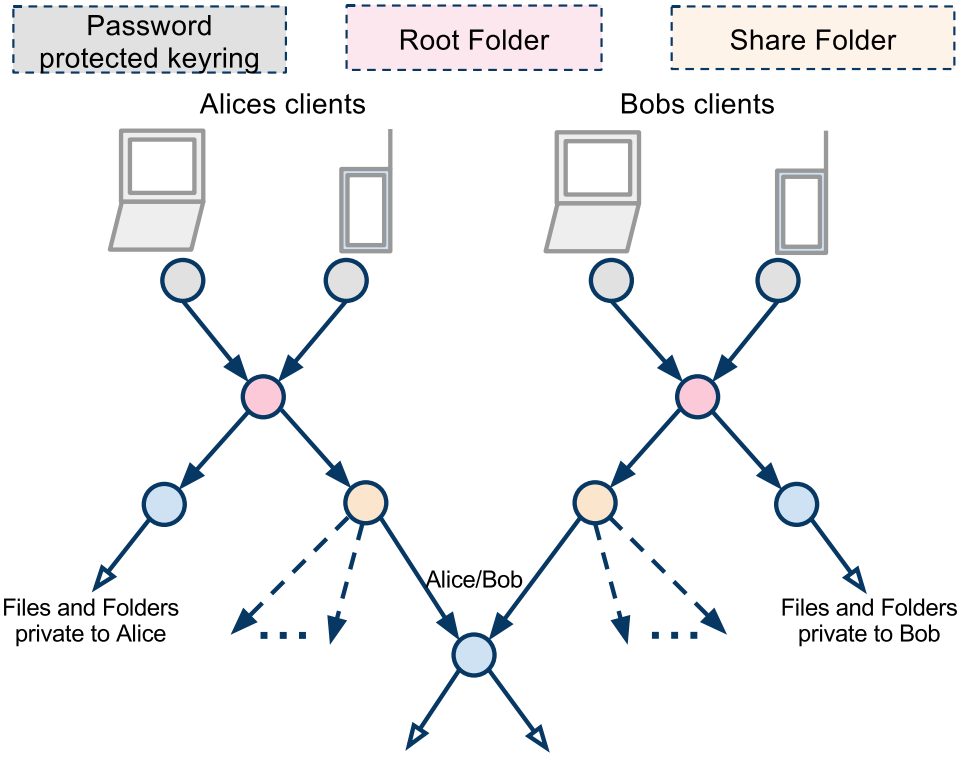

Figure 3.2: File system structure

3.1.2 ACL/Authentication Layer

The possession of a capability gives a user access to read a file or read or write a folder, and hence serve 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, accountability and possibly more access control could 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, based on who a user has shared a file with. This does however require the client to notify the server every time it shares a file object with a new person, and is strictly not necessary from a security standpoint.

Modification and deletion of objects

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 delete a folder.

For immutable files, there is no concept of write-access, only read. It is both illogical and impractical to assume that read access should also yield delete access. 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 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 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 for this file through the capability

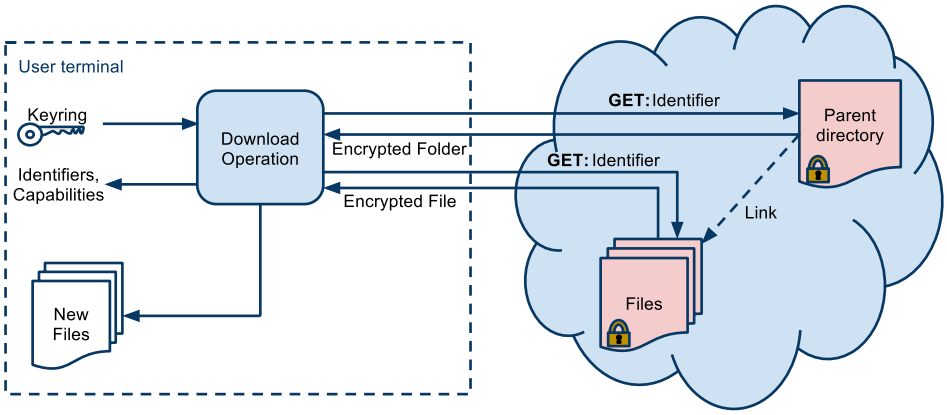

Figure 3.3: Scenario: Downloading a file

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

Upload File

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

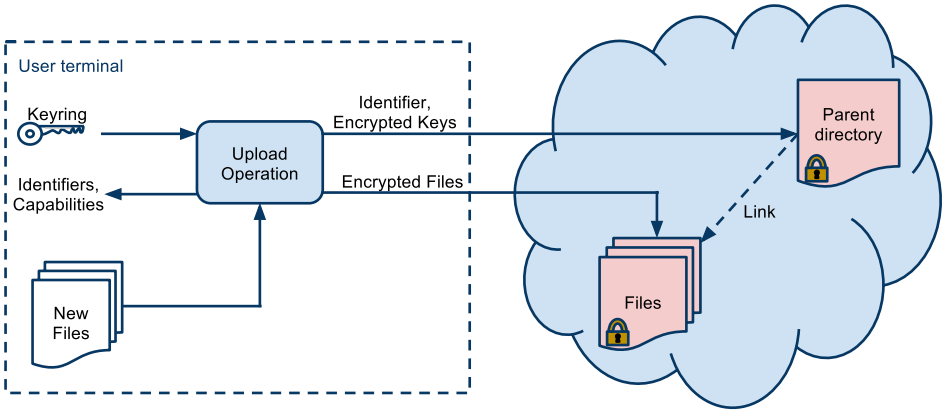

Figure 3.4: Scenario: Uploading a file

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.

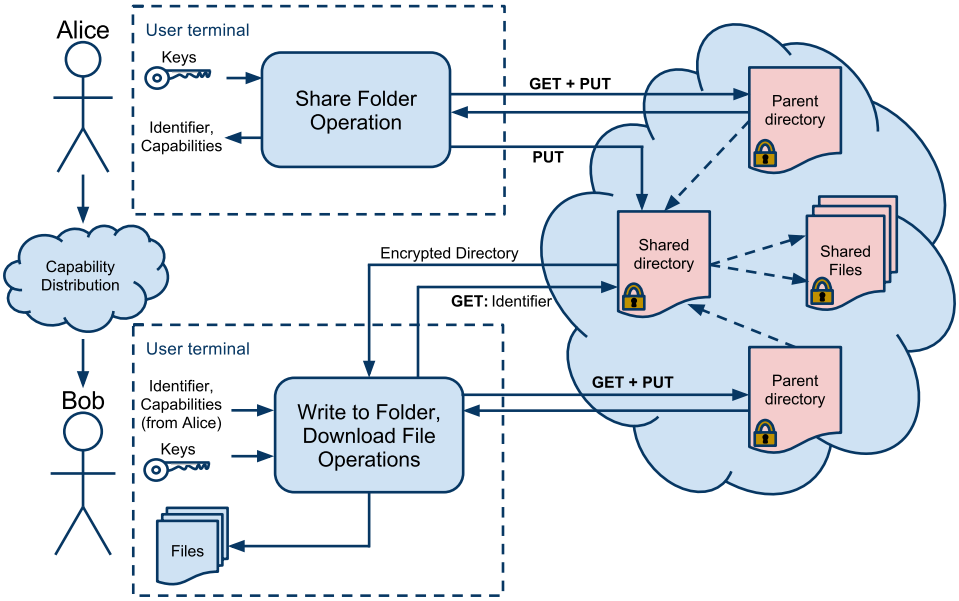

Figure 3.5: Scenario: Sharing files

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 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 6.2.2.

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 can not 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 find the directory another place 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.

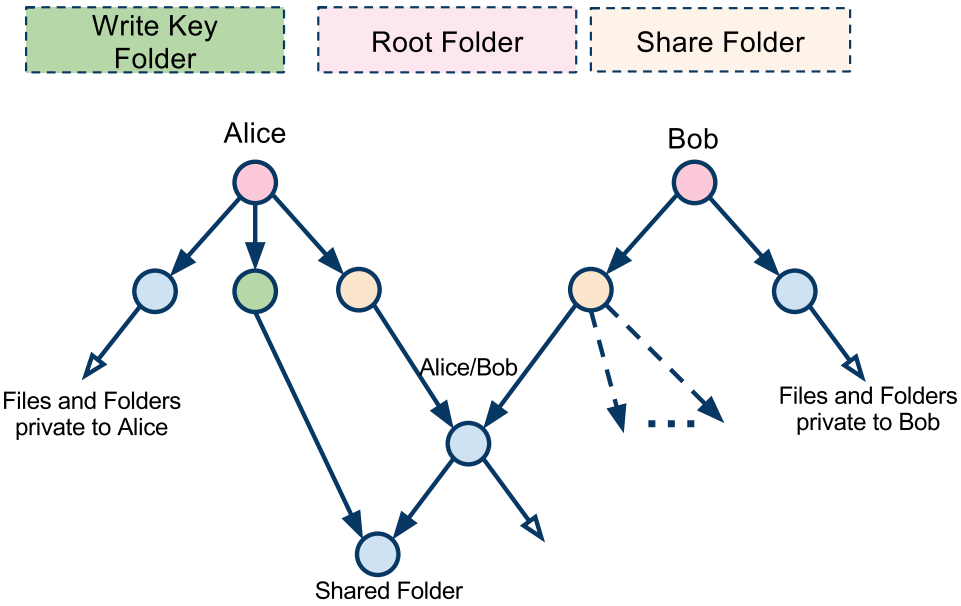

Figure 3.6: Sharing write-protected folders

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. These considerations will be brought up throughout the further description of the software.

3.2 Cryptographic Architecture

This section elaborates on the cryptographic solutions applied to the architectural scheme in Section 3.1. It will take a closer look at how confidentiality and integrity can be integrated into the proposed architecture.

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

We will start with a brief introduction explaining the fundamental security concepts utilized by the cryptographic architecture. The cryptographic architecture is further described in terms of file and directory operations. Key concepts are based on equivalent operations found in the Tahoe-LAFS [6].

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 files locally at the user's terminal before uploading them to the third-party storage provider.

When accessing a file, the application first downloads the encrypted file before decrypting it locally. To enable this simple encryption scheme, a user's terminal is required to locally 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 contents is summarized in Table 3.1, and will vary somewhat for files and folders, since they are implemented by respectively immutable and mutable files.

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 [24]. 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 [25].

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* and *open 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, any Message Autentication Code (MAC) could be used instead of HMAC-SHA1, and any cryptographic hash function could be used instead of SHA-256.

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, and is transfered 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 from the user terminal, is 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.

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

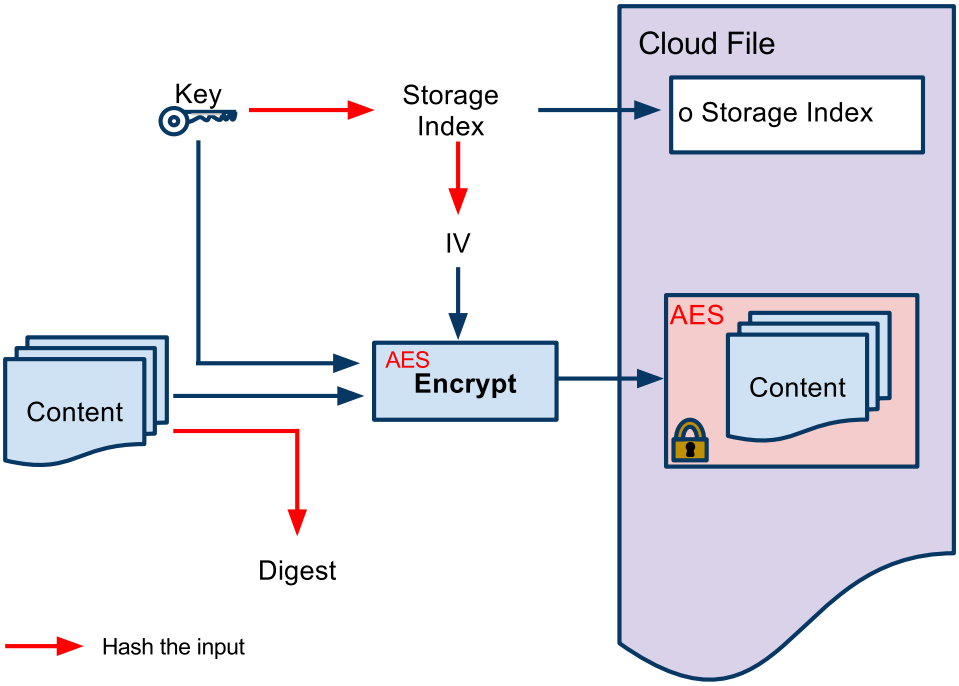

Figure 3.7: Behind the scenes: Uploading a file

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 even once more to form the *storage index*.

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 made from the write key with a MAC function and sent alongside the file. 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 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. If the write key is found, it can be hashed to obtain the read key. The read key is again hashed to form the storage index, which is used to request content from the server.

From the server, the user receives the encrypted contents of the folder, the

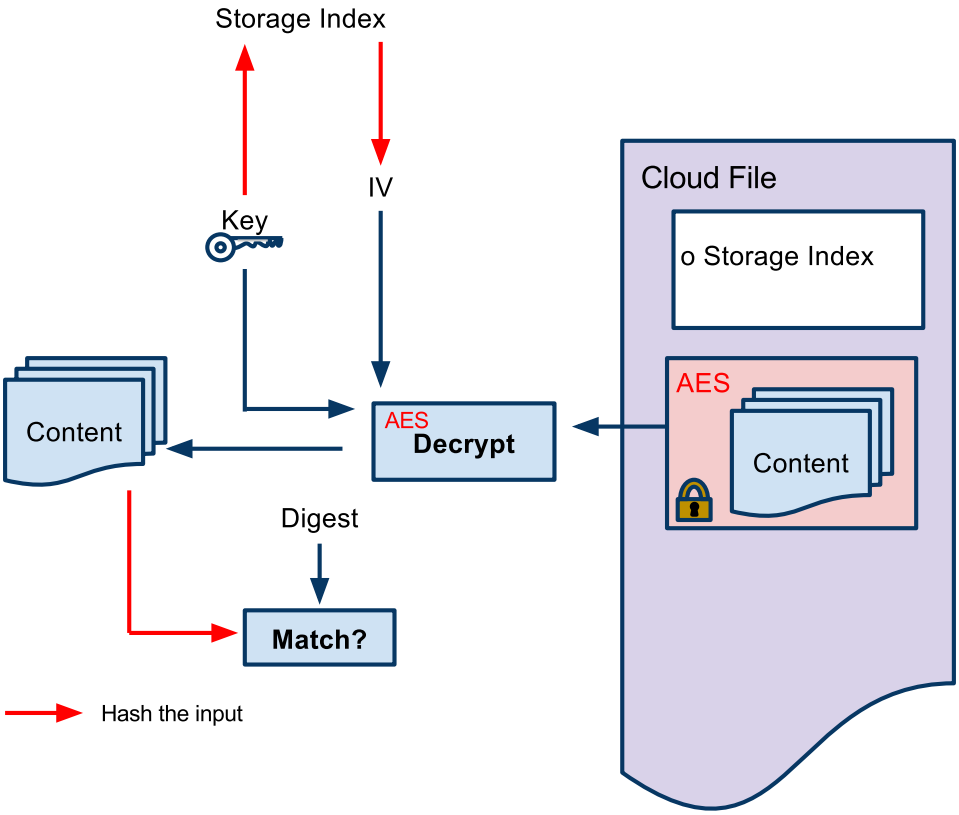

Figure 3.8: Behind the scenes: Downloading a file

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.

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 has the write key in a capability, and must use this to decrypt the encrypted signing key which he downloads from 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 fresh 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.

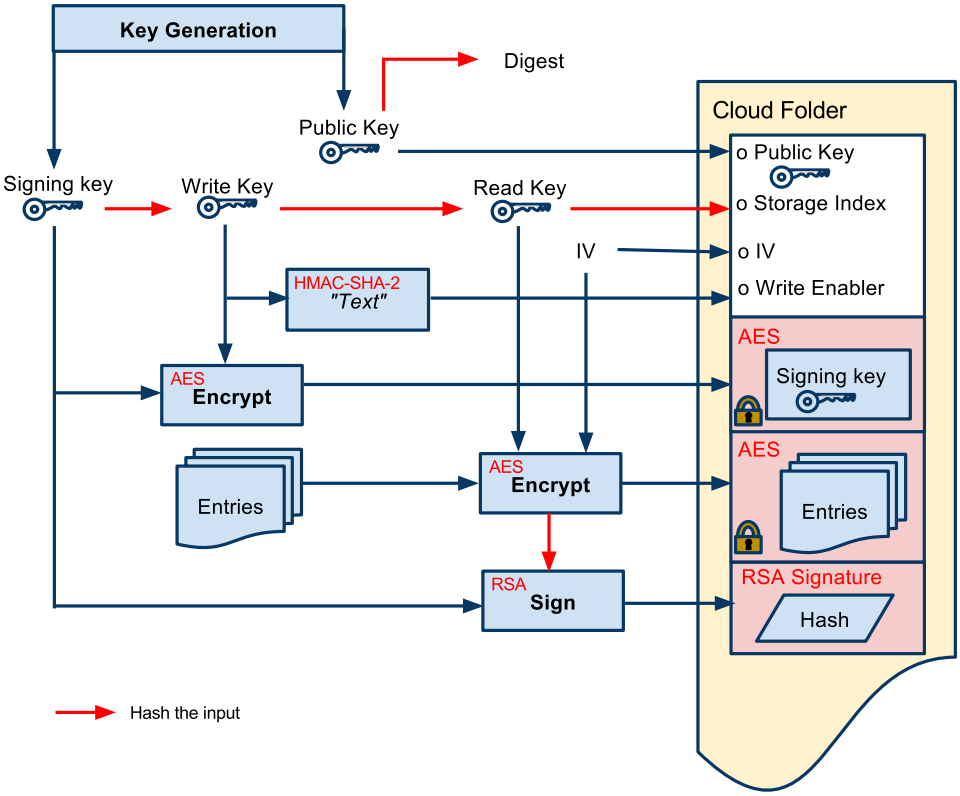

Figure 3.9: Behind the scenes: Creating a directory

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 signatures, as found 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 [26] compares studies listing predictions on how long primitives will be secure based on different sources with different predictions.

Symmetric Cipher

For a symmetric cipher the choice is pretty simple – AES. AES is the NIST standard for symmetric encryption and is readily available in most programming languages. We have chosen a key size of 128-bit. ECRYPT II [27] has one of the more pessimistic predictions on how long this will be secure, saying data encrypted

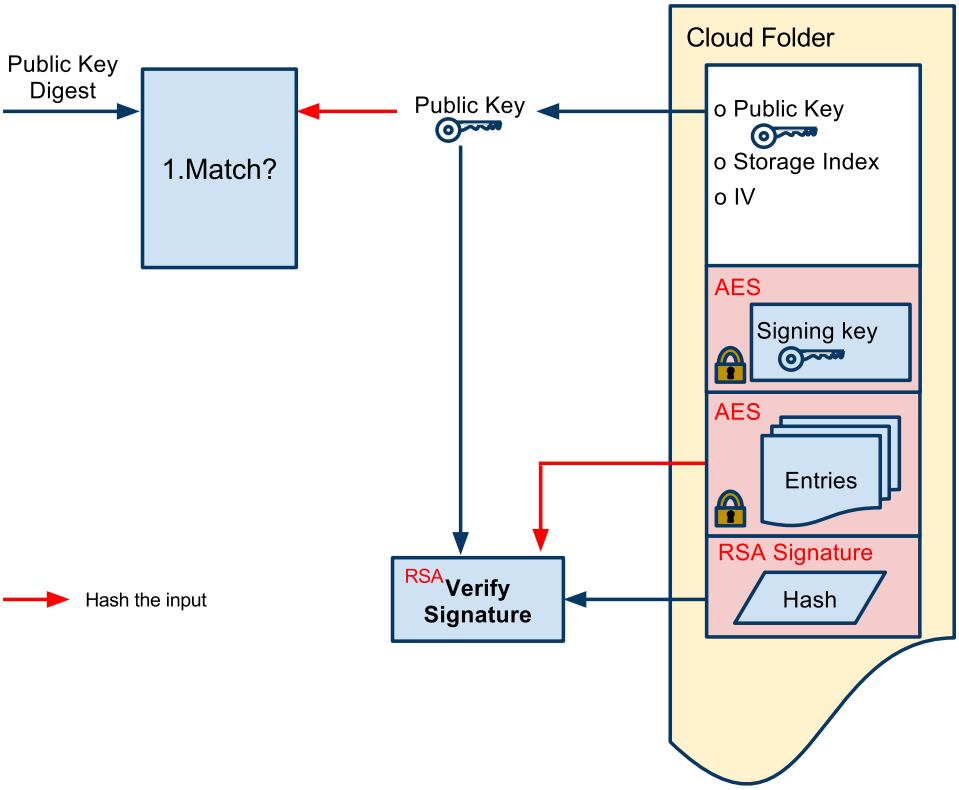

Figure 3.10: Verifying a directory

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.

Mode of operation The mode of operation we have chosen is CBC. This is based more on practical advice than on security consideration. Ferguson et al. [14] advices the use of either CBC or Counter (CTR) mode, where CBC mode is easier to implement correctly, while CTR mode, at least in theory, leaks the least information.

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. We decided on *PKCS5Padding* in our implementation, based on its existence in the cryptographic libraries we utilized.

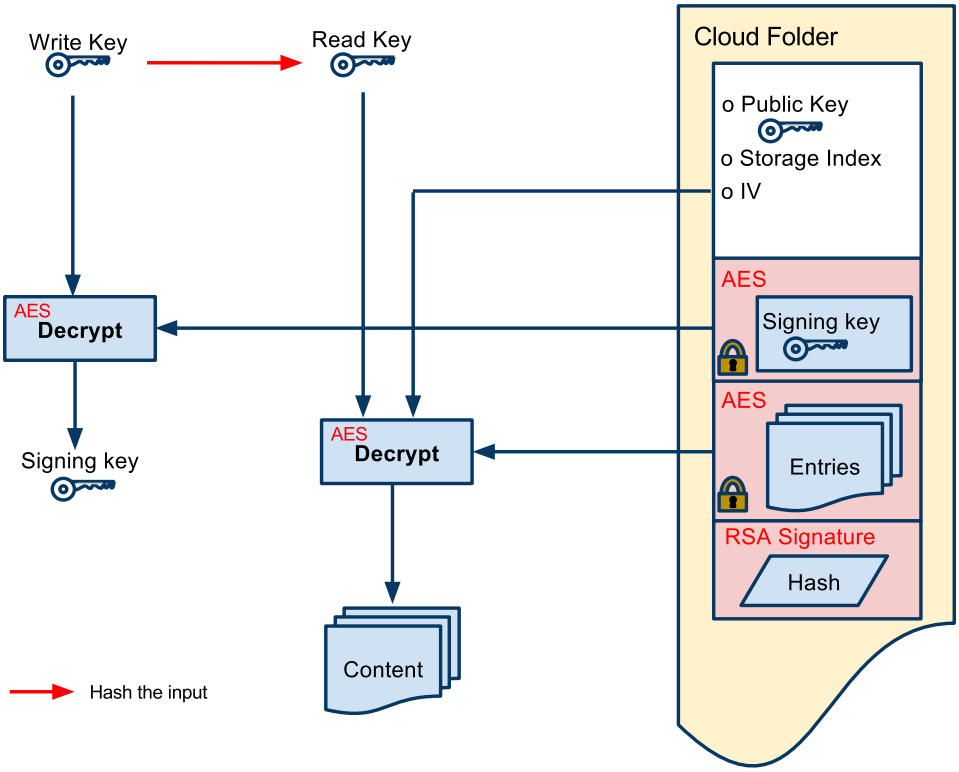

Figure 3.11: Decrypting the contents of a directory and obtaining the signing key

Cryptographic Hash Function

The SHA-family is the current standard for cryptographic hash functions, and from this we chose double SHA-256. The cryptographic scheme requires the hash 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 (e.g. by Christophe De Cannière and Christian Rechberger [28]) and 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 standard seems to be either RSA or DSA. Both functions would work, but we went with RSA, primarily because Tahoe-LAFS made the same choice.

There might however be a performance bonus in selecting RSA. An internet draft [29] 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 [30] 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 signing, updating and creating folders.

For the key length we have implemented 1024 bit, but in retrospect this is a bad choice. All sources at this point recommends at least 2048 bit.

MAC

The use of the MAC function in CSV is somewhat special. Ferguson et al. [14] defines a MAC to be a construction that detects tampering with a message – i.e. it authenticates the message. As depicted in Figure 3.9, the result of the MAC function is another key. By presenting this key, a user verifies *to the server* that he knows the write key.

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 origin key. On the basis of this, Ferguson et al. [14] recommend HMAC-SHA256.

Password Based Key Derivation

3.3 Server Implementation

The server, in the most basic form, has to support two operations – sending and receiving files. In addition, an Access Control List (ACL) layer is needed to support user management and access control to able to allow the deletion of files from the server.

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, it is well tested, will pass through most firewalls and has a multitude of libraries in programming languages. To get a working server we can also use any existing web server as a foundation, which saves a lot of work. Thus HTTP was chosen as our communication protocol.

REST

The Web is built around an architectural style called Representational State Transfer (REST) [31, 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 [31]. Our server application adheres to these constraints or *patterns* as follows:

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

Client-server
The server is our server application, and the client is the various client applications,

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 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.

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. When a folder is uploaded a Write Enabler must also be provided together with the storage index..

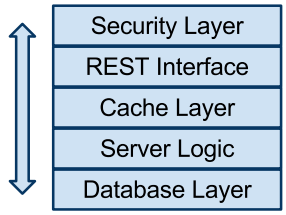

Figure 3.12: Architectural layers in the server application.

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 most basic form of MITM attacks. It also provides protection against eavesdropping, which would have revealed the Write Enabler for folders and could be used by an attacker to delete user files. 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 [32], 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 (i.e. `mod_wsgi`) 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 a micro framework around WSGI called Pyroutes[4]. It provides shortcuts 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

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

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.

```
|-- 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

```
1  from pyroutes import route
2  from pyroutes.http.response import Response
3
4  from cloudstorage.filesystem import (retrieve_file,
5                                       FileSystemException)
6
7  @route('/get')
8  def get_file(request, storage_index=None):
9      if storage_index is not None:
10         try:
11             file_to_send, size = retrieve_file(storage_index)
12         except FileSystemException, e:
13             return Response(e.text, status_code=e.code)
14
15         headers = [('Content-Type', 'application/octet-stream'),
16                    ('Content-Length', str(size))]
17         return Response(file_to_send, headers)
18
19     return Response('No resource ID given.', status_code=400)
```

ACL functionality

The only ACL functionality implemented, is the server-side verification that a client has proper access to overwrite a file, e.g. when a client wishes to update a folder with new contents. We call this **Write-Enabler Verification**.

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

If a client tries to put a file with a Storage Index that already exists, the server replies with an error code if the client in addition does not provide a valid Write-Enabler Key.

3.4 Client Implementation - Android

The Proof-of-concept (PoC) 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 speed and memory. The point of making the PoC 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 has a network connections, and it is 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 also have good performance on any faster device as well.

3.4.1 Environment

The PoC 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 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` [33].

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.

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 implementation of the JVM can have different implementations of the cryptographic primitives, but the developer does not need to know which implementation is available.

ZXing Barcode Scanner ZXing Barcode Scanner 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 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.

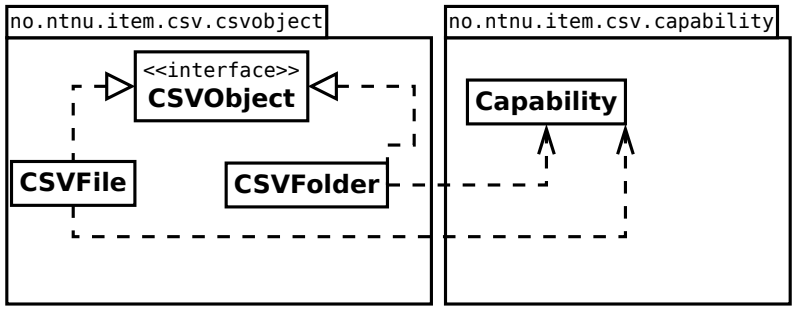

Figure 3.14: Cryptographic entities and their relations

Capabilities Capabilities are containers for cryptographic keys and information to identify a corresponding object. A capability 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 it's serialized form shown in Figure 3.15. *Object Type* specifies if the capability represents a file or a folder, with values F or D respectively.

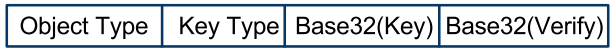

Figure 3.15: Serialized form of a capability

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

The different parts are separated by the 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 space the RAM on the device itself represents.

To keep the memory footprint low, we use the pipe-and-filter architectural pattern to stream data all the way from the cloud to the disk, 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

```
1  FileInputStream is = new FileInputStream("/file/path");
2  BufferedInputStream filebuffer = new BufferedInputStream(is);
3
4  MessageDigest md = MessageDigest.getInstance("SHA-256");
5  DigestInputStream digestInputStream = new DigestInputStream(
6          filebuffer,md);
7  BufferedInputStream digBuffer = new BufferedInputStream(
8          digestInputStream);
9
10 Cipher cipher = Cipher.getInstance("AES/CBC/PCKS5Padding");
11 // cipher is also inited with a random generated key, and an
12 // IV which is the digest of the key
13 CipherInputStream cipherInputStream = new CipherInputStream(
14         digBuffer, cipher);
15 BufferedInputStream readBuffer = new BufferedInputStream(
16         cipherInputStream);
17
18 // HttpClient uploads data by reading from readBuffer
19
20 // The digest from the file just uploaded
21 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 whatever 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. Buffers are placed between each step of the stream for some performance increase.

Communication with Server

For HTTP transport we utilize the Apache Software Foundations HttpComponents Client[5] also known as HttpClient.

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

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 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 copying the key from one users client to another, and afterwards verifying 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.

The other possibility is to make use of a *Quick Response (QR) code*, which is a matrix barcode that can store information. What this means is 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.

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 transfered, 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 users wish.

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 to much for a user to remember, so the capability will have to be stored on the client. However, the client might be stolen or broken into by some means, and if the capability is stored in clear text, it is easily stolen.

We partially remedy this by the use of the PBKDF2 algorithm, which makes an encryption key from a password. This generated key is used to encrypt the root capability in a file stored on the client. We call this the encrypted key ring.

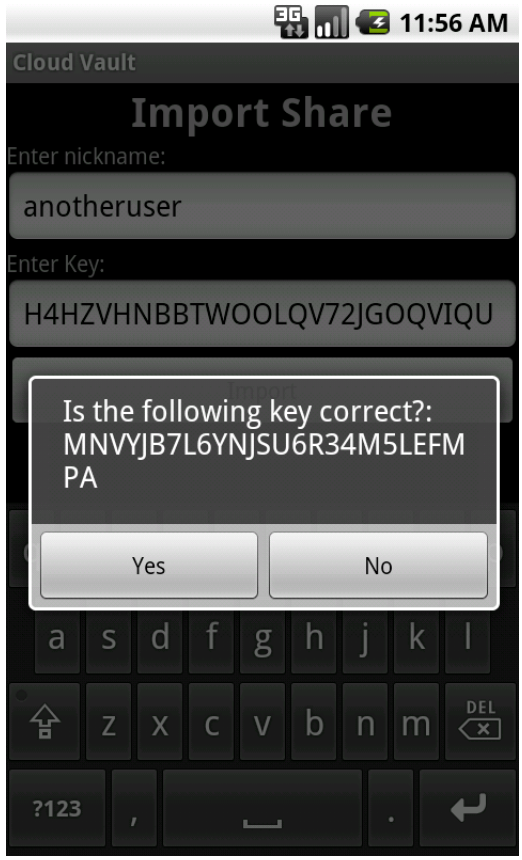

Figure 3.16: Establishing a share by copying the key

These precautions are however no defense if the attacker managed to read the memory while the key is unlocked. The client enforces that this password should be at least 9 characters long.

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. However, 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

Figure 3.17: Establishing a share by using barcodes

a password to encrypt the root capability in to the local keyring.. From the main screen the most common action would be to *Browse the vault* – in other words to see the files that the user has stored in the cloud.

Browse the Cloud The interface for browsing the 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.

An 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 filesystem 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 remove a file or a folder from the parent folder. The reason why it says unlink and not delete, is that a file can potentially be linked in a number of different folders, and it is impossible by our design to reveal which folders the file has been linked to, except the one that we are unlinking from.

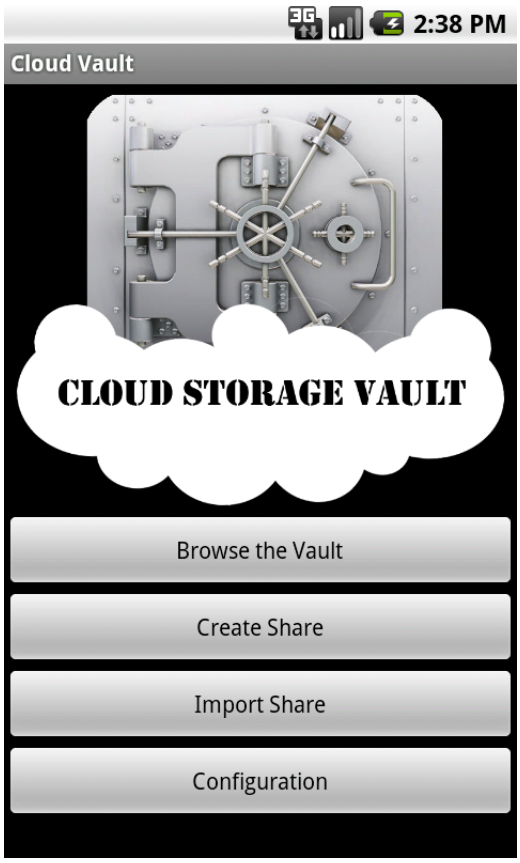

Figure 3.18: Main screen of the client application

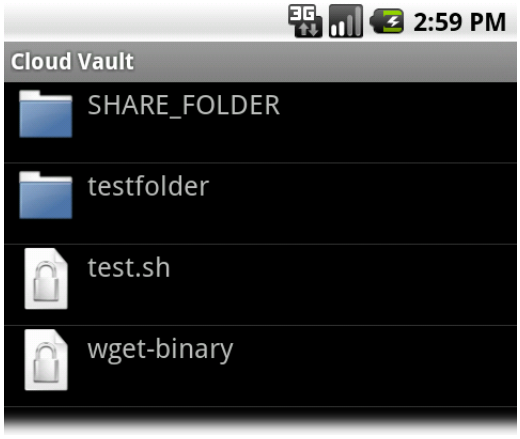

Figure 3.19: Browsing the cloud storage from the client

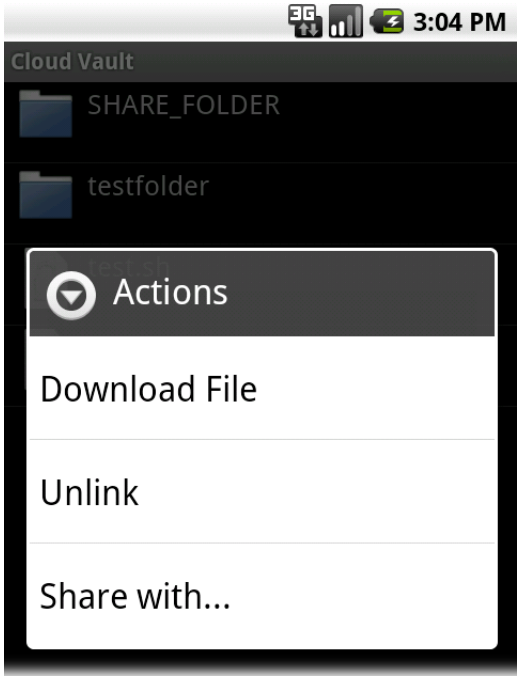

Figure 3.20: Context menu showing actions available for items stored

4

Experimental Procedure

This chapter will explain the experimental procedures performed on the Cloud Storage Vault. It will start by explaining the measurements on the performance of the application. The performance is further compared to similar existing network applications. Finally, the chapter will go through the experimental procedures of measuring some aspects of the security of the system.

4.1 Performance

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. We emphasis that the measurements are meant to be taken as indications of the performance of the scheme and implementation and not exact measurements. We also emphasis that the Android measurements are our main objective, since that is what the client is primarily

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

4.1.1 What is Measured

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

Folders are relatively small in size, and the implementation is not a pipeline. Compared to files, the heavy operations are done client side before the content is uploaded to the server. We therefore measure the speed of these operations, more specifically we measure:

1. The average time it takes to create a folder (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 How we Measure

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. We measure the average bandwidth from a few seconds after the file upload has been started, until the file upload is nearly complete. Granted this does not take overhead such as the initial

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

key generation into account, but for any file of some size this overhead should be neglectable.

To test the different folder operations we have to measure in the program code, we do this by the use of the Java function *System.currentTimeMillis()* [2], which we call before and after an operation and calculate the difference. When we test operations that are dependant on the contents of a folder we do this with folder entries of 86 Byte.

4.1.3 Eliminating Bottlenecks on Android Devices

On the Android devices, we identify three possible bottlenecks that we might be able to control: The *network*, our *application* by itself 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). While these protocols work just fine, from a measurement standpoint, we want to have a fast and stable connection. Our 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 [34]. 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.

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 facinating 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

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

[2] `http://download.oracle.com/javase/6/docs/api/java/lang/System.html#currentTimeMillis()`

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.

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.

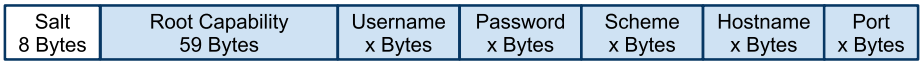

Figure 4.1: The keyring format with encrypted fields shaded in blue

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.

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 strenght. 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 BFDA, was designed for a single computer, while the second program, named 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[3] installed. It also depends on Java being configured to use Bouncy Castle as its primary JCE provider. These prerequisites are required because the

[3]Policy files enables Java to use stronger encryption than what is allowed by US export law.

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 [35], and the guide we used for configuring Apache Hadoop in a Cluster using Ubuntu Server, are made by Noll [36].

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 Elastic Compute Cloud (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 will present the quantifiable numbers retrieved when doing the experimentation in Chapter 4. This includes performance measurements on the proof of concept client, and figures on the attacks presented that target the local keyring.

5.1 Client Performance

This section presents the results of benchmarking of the proof of concept client, and highlights the most important results.

5.1.1 Files

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

We observe that the Android devices performance for uploads is severely lower for uploads than downloads, and that this is not the case for the computer. We also observe that the speed with encryption and hashing enabled is severely higher for all three devices. From the individual operation results for HTC Desire we note that the encryption and decryption part is the 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. The Tables 5.5 and 5.6 shows the time it takes to serialize and encrypt and sign the folder respectively. These two actions are what has to be performed every time a folder is changed, while the creation of 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 is operations that lead to an Out of Memory exception during execution.

We observe that that the slow 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 our implementation struggle to handle folders with large contents on the HTC Hero.

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.

Table 5.3: Speed of individual operation 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

However, the CDA attack does only support dictionary attacks. The results for both programs are given below.

5.2.1 Brute Force and Dictionary Attack (BFDA)

With BFDA we executed both a brute force and a dictionary attack to estimate the amount of passwords we could brute force within a second. The brute force and dictionary attacks achieved approximately the same results. However, for each measurement, the brute force attack turned out to achieve about 30 passwords more per seconds than the dictionary attack. This is believed to exist due to the file reading overhead in the dictionary attack. The results for both attacks are illustrated in Table 5.8 and Figure 5.1. The histogram in Figure 5.1 shows how many passwords per second we were able to brute force versus the different amounts of iterations used in PBKDF2.

Table 5.8: Speed results of running BFDA. Results are given for different numbers of PBKDF2 iterations. Speed results are measured in passwords checked per second.

PBKDF2 iterations	Brute force attack	Dictionary attack
500	3464.45	3451.55
1000	1812.01	1781.11
2000	926.53	883.78
4000	472.44	438.83

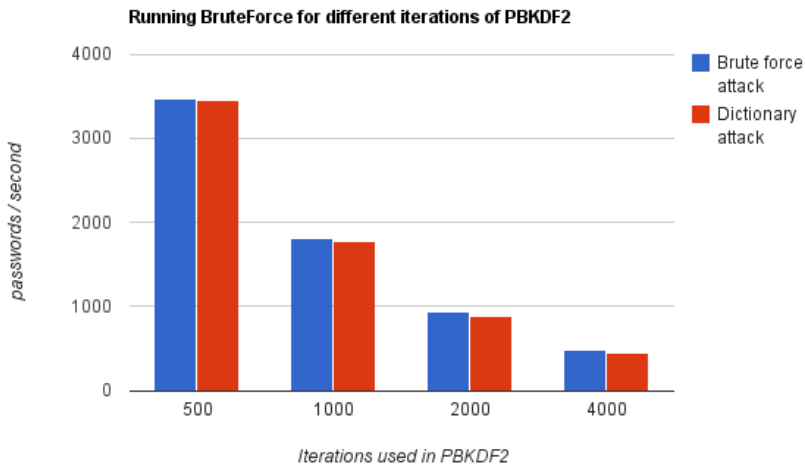

Figure 5.1: Results from running brute force and dictionary attacks against a local encrypted keyring.

Figure 5.1 indicates that by doubling the number of iterations in *PBKDF2*, the efficiency of a brute force attack would decrease by half of it's value.

5.2.2 Cluster Dictionary Attack (CDA)

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 a small difference where the local computer achieved about 100 passwords per second more than each instance in the cluster. With respect to difference in CPU power between the local computer and a single cluster instance [37], this is an expected result. With 20 nodes in the cluster we were able to brute force around 35 000 passwords per second, given *PBKDF2* with 1000 iterations.

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 the combination of symmetric keys derived from asymmetric keys, the scheme supports safe sharing of folders in both read-only and read-write modes.

Cryptographic Primitives A significant design difference, is that it is substantially more difficult to change cryptographic primitives for the user in Tahoe-LAFS than in CSV.

6.1.2 Sharing

One of the major advantages of CSV over other 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 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 sharing of files and directories implicitly deals with the general problem of key distribution. The cryptographic scheme does not propose specific solutions for this, and leaves it up to the implementation to choose the best suiting method in light of the specific requirements. The procedure used in the proof of concept system developed, are described in Section 6.2.

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/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 as well as folders.

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 reach zero, the user notifies the server to delete it.

Loop Detection in a Directed Acyclic Graph

In remedy of 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.

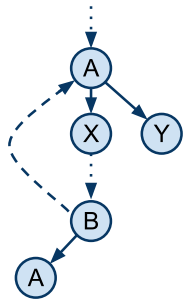

Figure 6.1: Theoretical cycle in the directory graph

Tarjan's Algorithm [38] 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 pretty much every way a user wants to use an ordinary file, with the exception of the case where the user wants the entire file and the file has not been tampered with.

The problem with the scheme, is that the user will have to download the whole file, before he can verify by absolute certainty 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.

If the file is a movie file, and the user only wants to look at a small part of this file, this is not possible with the current verification scheme. There is no way for the user to verify only parts of a file, and he will have to download the whole file, which is suboptimal.

A possible solution to support this kind of streaming, would be to build a *hash tree* of the whole file, as shown in Figure 6.2. With a hash tree, smaller bits 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 ciphertext, and the capability could hold information to verify the hash 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 the Tahoe-LAFS [6].

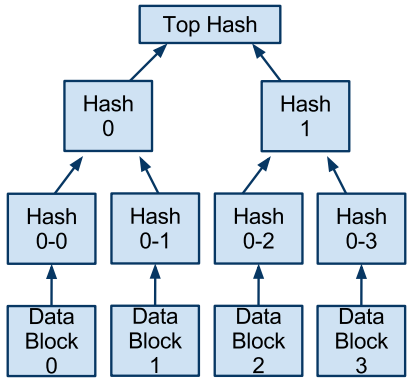

Figure 6.2: Hash tree of a file

6.1.5 Version Control System

Some of the currently available cloud storage systems, as Wuala and Dropbox mentioned in Section 2.7, 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. This could be implemented in CSV as follows.

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.

Type of Mutable File A new form of mutable file could be implemented in practically the same shape 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 secure cloud 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 movie 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 The 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* [6].

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 an extra 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 duplication 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.

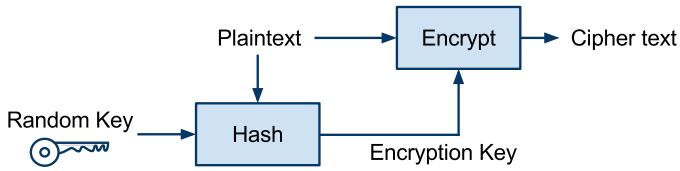

Figure 6.3: Tahoe-LAFS deduplication scheme

6.2 Implementation

This section will discuss and deal with the different choices taken to fulfill the implementation of the cryptographic scheme. We will start by discussing the use case we chose to implement. We will further analyse our choice of key distribution and look at alternative solutions. The section will end by discussing the most important cryptographic primitives chosen and utilized in our implementation.

6.2.1 Choice of Use Case

There are mainly two types of users for a cloud storage solution, personal or enterprise/organizational. What we have implemented is aimed more against the personal users than the enterprise market. The main difference between these two use cases is that in an enterprise there will usually be some kind of IT department which will have the possibility of setting up users terminal, which in turn means that the initial client setup can be more advanced. Another difference is that it would usually be more defined who you are supposed to share files with and what you should have access to. An organization might not approve of a model where a random user can forward read rights to other users.

6.2.2 Choice of Key Distribution

The challenge of key distribution is a hard one. To successfully authenticate a user and transfer a key requires either that you trust a third party, you meet a user in person or that you use some out of bound safe channel. Since our initial problem is that we do not trust the cloud provider we do not want to use a trusted third party either. This pretty much leaves verifying another user in person, although a scheme similar to PGP could be used. In PGP users can publish that they have verified other users and to which degree they trust them. Other users can use this information to calculate the probability that a certain user is legit. The reason why we do not implement this scheme is that it is fairly complex for a standard user. There are however nothing that stops a user that want to use PGP to transfer capabilities through it.

If our cryptographic scheme 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, who enforces that all certificates granted to users are correct. A user could then be simply prompted by the name of the user she would like to share a file with, and the rest could be handled by logic and the trust of the PKI.

6.2.3 Choice of Cryptographic Primitives

6.3 Performance

The measurements of our client reveal that it is not as fast as you would want it to be. For files, a perfect client would render the network or the SD-card as the bottleneck, but our measurements state that this is clearly not the case. The speed we get when downloading or uploading a file is decreased by about 50-80% on all three devices. The results for the individual operations performed on HTC Desire shown in Table 5.3 clearly indicates that encryption and decryption are the slowest elements in the pipeline, and responsible for most of the performance drop. Another thing to note is that the implementations of InputStreams and OutputStreams in Java reads or writes the requested number of bytes stream by stream[1], so we will never reach even the speed of the slowest component in the pipeline. Splitting the pipeline between different threads should make it possible to reach higher speed in the pipeline, especially the computer which spots multiple processor cores.

Folders For folders our application is performing quite good, if we look at only the cryptographic options. Here the slowest operation is clearly the generation of new cryptographic keys. For a user of our 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.

The other cryptographic operations on folders runs quite quickly, but the performance for serializing the folder is painfully low when folders reach a large size. Now a folder with thousands of children is not something we expect a 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 reason why this operation is to be blamed on our implementation. The contents are implemented as a hashmap, which is serialized to a string to make it ready for encryption. Ideally this step is not necessary, if the contents exists in the correct form in memory by default. Even if this might not be possible for other reasons, such as decreased performance when manipulating the folder contents, we believe a faster implementation is possible.

[1] `http://download.java.net/jdk6/source/`

6.3.1 Accuracy of Measurements

6.4 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.

This section will discuss the general security provided by the Cloud Storage vault. The security of the Cloud Storage Vault is 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 data encrypted on the server, and also know whether a stored data object corresponds to a file or a folder.

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. In other words, a folder is attackable both trough the asymmetric and the symmetric cipher used. If the confidentiality of a folder is breached it will additionally enable the attacker to decrypt all of the corresponding sub-folders 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.

Disclosure of Keyring Content

The confidentiality of the cryptographic root key and user passwords is held by a single encrypted keyring that is stored locally for each user client. The keyring is encrypted with a cryptographic key derived from a user password. Unfortunately, using a password to encrypt the keyring, makes the keyring a suitable target for brute force and dictionary attacks. The keyring's vulnerability against brute force and dictionary attacks was experienced in Section 4.2.

Brute Forcing the Keyring The confidentiality of the root folder can be breached by brute forcing a user's locally stored keyring. This was experienced in Section 4.2. From the results in Section 5.2 and Figure 5.1, 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 half of it's value.

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

With the password requirements above and with 4000 iterations in *PBKDF2*, it will be extremely hard to perform a plain brute force attack against the local keyring.

Even though the keyring is secured against a plain brute force attack, it is still vulnerable to slow dictionary attacks. To completely avoid brute force and dictionary attacks against the keyring, the most efficient and preventive action would be to simply ensure that the encrypted keyring is only accessible by its authorized users.

The Weakest Link

6.4.2 Client Security

Given enough users, sooner or later, there will be a user who looses his client. Additionally, a user's client might also be broken into without the user's knowledge. This section will look at the application's security in both scenarios and further discuss the security of the encrypted keyring stored inside a user client.

Loosing the Client

If a user looses his client and the client is logged on to the user's Cloud Storage Vault, anyone who finds the client can get access to the user's private files and folders. While logged on to the service, it is also possible for an experienced attacker to reveal the user's server access password as well as the password to decrypt the locally stored keyring. The attacker can achieve the passwords by scanning the client's memory while it is logged into the Cloud Storage Vault. Although this is a severe security breach, it can easily be avoided by an additional mechanism to the implementation. The Cloud Storage Vault can simply flush the password values stored in memory right after a successful login session.

Even though this mechanism prevents disclosure of critical access data when loosing a client, it does not prevent a logged in attacker from disclosing the user's encrypted files and folders. To minimize this problem the Cloud Storage Vault should implement a timeout mechanism for inactive users. If a user has been inactive for a given amount of time, the user should be logged out of the application. In this way, the application will reduce the consequence of losing the client. It is important that the timeout is set to a value between what is acceptable by the users and what can be defined as secure timeout[2].

[2]Users would not like to write their password for every 30th second. Additionally, a timeout

Even though a timeout value may deny an attacker access to the storage vault, it can not prevent the attacker from accessing files that have already been downloaded from the Cloud. There is, however, two solutions to handle this problem. The first solution is to avoid that downloaded files are written to the client's storage medium. This can be implemented using a temporary filesystem in the client's memory[3]. However, using only the client memory to store files, will greatly limit the possible size of the downloaded files. In case limited file size is not an acceptable solution, it is possible to use a temporary filesystem that supports swapping. This will, however, keep parts of a large file in the client's storage medium, revealing information about a downloaded file. The most important feature with the temporary filesystem is that the downloaded files are only stored temporary, while they are opened by the application etc.. This will prevent an attacker from revealing previously downloaded files from the cloud.

It is important to mention that a temporary filesystem for downloaded files can not be used in case users want to store their downloaded files. A solution would be for the Cloud Storage Vault to store downloaded files in their encrypted form. The downloaded files could then only be opened locally through the Cloud Storage Vault application, which would further decrypt and send the file to the appropriate application. It is worth noticing that this mechanism will also enable the possibility of file synchronization between the client and the cloud, similar to the one of Dropbox.

Possessing a Compromised Client

As mentioned earlier, an attacker can also break into a user's client, using a backdoor, trojan horse etc.. Depending on the attack, a worst case scenario would enable the attacker to obtain root access on a user client. With root access, the attacker is given complete control and privileges over the client and can access whatever data resource desired within the client. With this in mind, the attacker can read, write and execute every file that has been downloaded by the Cloud Storage Vault. However, this can be prevented by using a temporary filesystem or storing downloaded files in their encrypted form as previously described.

Given root access, the attacker can also utilize software to read live content from the client's memory. In this case, it is unfortunately not possible to protect against disclosure of passwords or capabilities used in the Cloud Storage Vault. The Cloud Storage vault is dependent on storing the keyring password in memory during the login procedure. Capabilities are also stored in memory during browsing of contents in the cloud. Both passwords and capabilities are therefore retrievable by the attacker, even though their necessary storage time in memory is kept to a minimum by the application. Another unmanageable scenario would involve the attacker installing a keylogger on the compromised client. The keylogger can further detect the keyring password, when the user logs in to the application.

---

value of 30 minutes is not secure, as it is long enough for someone to posses the user's client.

[3] ramfs and tmpfs are examples of temporary filesystems for clients running Unix-like operating systems

It should be concluded that a compromised client, in a worst case scenario, will not be able to support secure usage of the Cloud Storage Vault.

6.5 Additional Features

7

Conclusion and Future Work

7.1 Compared with Other Solutions

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

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

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

[15] 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`.

[16] 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`.

[17] 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`.

[18] 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`.

[19] Trusted Computing Group. `http://www.trustedcomputinggroup.org/`. Retrieved 03.15.11.

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

[22] 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`.

[23] 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`.

[24] 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.

[25] 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.

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

[27] 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.

[28] Christophe De Cannière and Christian Rechberger. Finding SHA-1 Characteristics: General Results and Applications. In Xuejia Lai and Kefei Chen, editor, *Advances in Cryptology – ASIACRYPT 2006*, Lecture Notes in Computer Science, pages 1–20. Springer Berlin / Heidelberg, 2006. From http://www.springerlink.com/content/q42205u702p5604u/.

[29] 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.

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

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

[32] 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.

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

[34] 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.

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

[36] 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.

[37] CPU Benchmarks. http://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+E5410+%40+2.33GHz, 2011. Retrieved 05.10.11.

[38] 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.

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

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

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

Appendices

A

Other Relevant Implementations

This chapter will study implementation details from applications that were created, but not a part of the thesis' main topic. Applications considered are Brute Force and Dictionary Attack (BFDA) and the Cluster Dictionary Attack (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 [39].

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

```
1  public void bruteForceAttack(String[] input) {
2    current_word = new char[1];
3    words = new Stack<String>();
4    THREADS = Integer.parseInt(input[2]);
5    bf_threads = new BruteForceThread[THREADS];
6    MAX_WORD_LENGTH = Integer.parseInt(input[1]);
7
8    for (int i = 0; i < bf_threads.length; i++) {
9      bf_threads[i] = new BruteForceThread("bf" + i);
10   }
11
12   waitForBFThreads();
13
14   while (!found) {
15     if (current_word.length > MAX_WORD_LENGTH)
16       break;
17     pushWord(current_word.length - 1);
18     char[] tmp = new char[current_word.length + 1];
19     tmp[tmp.length - 1] = ' ';
20     System.arraycopy(current_word, 0, tmp, 0, current_word.↩
         length);
21     current_word = tmp;
22   }
23 }
```

`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

```
1  public void dictionaryAttack(String[] input) {
2    File dict = new File(input[1]);
3    THREADS = Integer.parseInt(input[2]);
4
5    if (dict.exists()) {
6      try {
```

```
7         fr = new FileReader(dict);
8         buf = new BufferedReader(fr);
9         start = System.currentTimeMillis();
10        for (int i = 0; i < THREADS; i++) {
11          new DictionaryThread("dict" + i);
12        }
13      } catch (FileNotFoundException e) {
14        e.printStackTrace();
15      }
16
17    } else {
18      System.out.println("ERROR: Dictionary file does not exist!↩
            ");
19      printHelp();
20      System.exit(0);
21    }
22  }
```

`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 filesystem HDFS, that can store data across multiple cooperating machines [40]. 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 [41].

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

```
1  @Override
2  public void map(LongWritable key, Text value,
3      OutputCollector<Text, LongWritable> output, Reporter ↩
           reporter)
4      throws IOException {
5
6    String[] line = value.toString().split(" ");
7    String[] line_chunk = new String[WORDS_PER_THREAD];
8
9    // Create threads to check words in line
10   for (int i = 0, c = 0; i < (THREADS * WORDS_PER_THREAD)
11       && i < line.length; i += WORDS_PER_THREAD, c++) {
12     if (line.length - i >= WORDS_PER_THREAD) {
13       System.arraycopy(line, i, line_chunk, 0, ↩
             WORDS_PER_THREAD);
14     } else {
15       System.arraycopy(line, i, line_chunk, 0, line.length - i↩
             );
16     }
17     dictionary_threads[c] = new DictionaryThread("dict" + i, ↩
           line_chunk);
18   }
19
```

```
20    // Wait for all threads to finish
21    for (DictionaryThread dt : dictionary_threads) {
22      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 behavior 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 CD-ROM.

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 to the proof of concept system, i.e. the background library, server and client.

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/`